UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Facoltà di Ingegneria
Corso di Studi in Ingegneria Informatica

tesi di laurea

Sviluppo di sistemi Linux embedded per applicazioni critiche.

Anno Accademico 2007/2008

relatore
Ch.mo prof. Domenico Cotroneo

correlatore
Ch.mo prof. Christian Di Biagio

candidato
Nicola Fragale
matr. 534/1853

A mia mamma e a mio papà per
la fiducia e la pazienza

Indice

Introduzione 5

Capitolo 1. Analisi del problema 8

1.1 Requisiti hardware e software 11
1.2 Sistemi operativi real time 14
1.3 La soluzione Dual Kernel 21
1.3.1 RTLinux - RealTime Linux 22
1.3.2 RTAI - Real Time Application Interface 23
1.4 La soluzione di MontaVista 24
1.4.1 Preemption patch 24
1.4.2 Low Latency patch 25
1.4.3 Scheduler O(1) 26
1.5 MTD e Journaling Flash File System 29
1.6 La Cross Compilazione 35
1.7 Dal bootstrap al login 37
1.7.1 Il bootloader 38
1.7.2 L'inizializzazione del kernel 42
1.7.3 Init, il padre di tutti i processi 44
1.8 Ottimizzazione dello spazio 49
1.8.1 uClibc 50
1.8.2 BusyBox 51

Capitolo 2. Embedded Finmeccanica Linux 55

2.1 Costruzione della toolchain 57
2.1.1 BuildRoot 57
2.1.2 Patch e configurazione del kernel 63
2.1.3 Gentoo 69
2.2 Preparazione delle partizioni 73
2.3 Immagine del root filesystem 76
2.3.1 Installazione del bootloader 77
2.4 Implementazione della distribuzione 78
2.4.1 Installazione del Kernel 79
2.4.2 NtpClient 79
2.4.3 Bash 80
2.4.4 Gdb 81
2.4.5 BusyBox 81
2.4.6 OpenSSh 83
2.4.7 Glibc 84
2.5 Risoluzione delle dipendenze ed altro software d'utilità 86
2.5.1 Zlib 86
2.5.2 OpenSSL 87
2.5.3 Ncurses 87

2.5.4 Kbd 89
2.5.5 Mtd-utils 89
2.5.6 Lzo 89
2.6 Configurazioni e installazione 90
2.6.1 Montaggio dei filesystem 90
2.6.2 Inizializzazione del sistema 92
2.6.3 Immagine su filesystem jffs2 95

Capitolo 3. Sviluppi 98

3.1 Struttura del filesystem 98
3.2 Filesystem di EFML 100
3.2.1 Spazio occupato dal filesystem di EFML 101
3.3 Occupazione di memoria del kernel 106
3.4 Tempi di boot 110
3.4.1 TSC – Time Stamp Clock 110
3.4.2 Bootchart 113
3.4.3 Upstart 114

Capitolo 4. Test 115

4.1 Gcov 116
4.2 Linux Test Project 117
4.2.1 Setup della suite LTP 118
4.2.2 Test delle caratteristiche real time del kernel 122
4.2.3 Test del sistema operativo 154

Capitolo 5. Conclusioni 190

Appendice A – Dipendenze del software 193
NtpClient 193
bash 193
Gdb 193
BusyBox 194
OpenSSh 194
Kbd 197
Mtd utils 200

Appendice B – Test 205

Bibliografia 210

Introduzione

Con il termine sistema embedded si identificano genericamente dei sistemi elettronici a microprocessore progettati appositamente, sviluppando hardware ad hoc, per una determinata applicazione e inseriti nella macchina della quale dovranno controllare e gestire tutte o parte delle funzionalità. A differenza dei comuni computer general purpose i sistemi embedded essendo dedicati sono stati costruiti riducendo l'hardware e i dispositivi necessari all'essenziale, con conseguente riduzione sia dello spazio occupato che dei costi e dei consumi. Il primo sistema embedded moderno fu il sistema di guida dell'Apollo, installato sia sul Lem che sull'orbiter. Il primo sistema embedded prodotto in massa fu la guida di un missile. Oggi questi sistemi pervadono la nostra vita, sono utilizzati praticamente in tutti gli apparati elettronici di uso comune e non. Sono presenti in svariati elettrodomestici, nelle televisioni, negli sportelli Bancomat, nei lettori cd/dvd e mp3, nelle centraline elettroniche installate a bordo delle automobili, nei set top box, nei decoder, nei telefoni cellulari, nei satelliti per le telecomunicazioni, nei router, nei rover inviati ad esplorare Marte. A causa della loro natura dedicata, in passato si è privilegiato l'uso di processori aventi la capacità di calcolo minima necessaria al compito da eseguire. I progressi compiuti dall'elettronica negli ultimi decenni hanno permesso di incrementare sia la capacità di calcolo che la velocità dei processori, contemporaneamente le memorie sono diventate sempre meno costose e più capaci. Tutto ciò ha di fatto contribuito a diminuire le differenze tra i sistemi genaral purpose e i sistemi embedded, permettendo su quest'ultimi l'uso di software sempre più complessi. Un parametro particolarmente importante per i sistemi embedded è l'efficienza. Questi sistemi devono spesso funzionare ininterrottamente senza intervento esterno e in ambienti estremi per anni. La necessità di garantire lunghi periodi di funzionamento e la riduzione del costo delle memorie a stato solido, ha favorito la diffusione e l'uso di memorie flash al posto degli hard disk, eliminando gli elementi con parti meccaniche in movimento più soggetti a malfunzionamenti e rotture. Una volta in esercizio, alcuni di questi sistemi possono

diventare fisicamente inaccessibili, come ad esempio i satelliti, è quindi necessario che siano in grado di resettarsi autonomamente in conseguenza a malfunzionamenti. Le crescenti necessità di calcolo e prestazioni dei sistemi embedded, unite alla disponibilità di hardware sempre più performante e dai costi sempre più contenuti, hanno portato all'uso al loro interno di completi sistemi operativi, spesso derivati da quelli normalmente utilizzati su normali pc desktop o server. Oggi sono utilizzabili sistemi operativi come VxWorks, Windows CE, QNX Neutrino, Linux. La costruzione di un sistema embedded e del software che lo equipaggerà è una operazione complessa che necessita l'analisi di diversi aspetti. La particolare natura dei compiti che i sistemi embedded devono eseguire rende necessario l'uso di sistemi operativi real time, i quali hanno la peculiarità di essere soggetti a vincoli temporali molto stringenti, dovendo garantire la risposta ad uno stimolo esterno in tempi certi. Un altro degli aspetti caratteristici dei sistemi embedded è il modo in cui il software che sarà installato su di essi viene sviluppato. Il kernel, il software di amministrazione e gli applicativi che andranno a completare il sistema operativo sono realizzati con uno sviluppo cross-platform. Il software è sviluppato su di una piattaforma dotata di un certo hardware, equipaggiata con un determinato processore, un sistema operativo e una toolchain (compilatore, debugger, librerie), detta sistema host, per un'altra piattaforma, detta sistema target e dotata generalmente di un hardware dedicato e di un processore diverso da quello montato sull'host. Il software principale che permette lo sviluppo cross-platform è il cross-compilatore, un compilatore che è eseguito sulla macchina host e che produce codice eseguibile per il processore della macchina target. Altri aspetti da esaminare per la creazione di un sistema embedded riguardano il filesystem da utilizzare, il bootloader, il kernel, il software per l'amministrazione del sistema, la shell e gli applicativi utente. Il root filesystem dovrà avere dimensioni contenute per poter essere installato agevolmente sulla memoria a stato solido che equipaggia il sistema embedded. Una volta creato il root filesystem è necessario trasferirlo dalla macchina host alla macchina target. Gli ultimi passi da compiere riguardano la configurazione del bootloader, affinché possa caricare il kernel, e degli script di

inizializzazione del sistema.

Lo scopo di questo lavoro è quello di realizzare una distribuzione Linux embedded per Finmeccanica, atto a girare sulla scheda VP417 prodotta dalla Concurrent Technologies. É richiesto che il sistema sia bootabile da una memoria flash presente sulla scheda, che il kernel da utilizzare sia Linux esteso con apposite patch real time. Nel primo capitolo si è esaminato lo stato dell'arte nella realizzazione di sistemi embedded basati su kernel Linux e software libero. Quali sono le ragioni che dovrebbero spingere sia un'azienda che un progettista di sistemi embedded ad utilizzare il software libero per la realizzazione del loro prodotto; cosa sono i sistemi operativi real time e quali soluzioni sono state adottate per trasformare il kernel Linux in un kernel real time; cosa offrono le memorie a stato solido e come è possibile utilizzarle in modo efficiente; come sfruttare nel modo più efficiente lo spazio disponibile sulla flash, selezionando e configurando in modo opportuno i software e le librerie che andranno installati sulla macchina target. Quali sono le varie fasi che interessano il boot della macchina, come è strutturato il filesystem. Cos'è e perché è utile la cross compilazione. Nel secondo capitolo sono descritte le operazioni compiute per realizzare la distribuzione, dalla configurazione e compilazione del kernel e dei software aggiuntivi, alla creazione del root filesystem. Sono state realizzate due distribuzioni, la prima basata sulle librerie C glibc, la seconda basata sulla libreria uClibc e costruita con il tool buildroot. Nel terzo capitolo sono raccolte le metriche della distribuzione basata sulle glibc. Quali strumenti sono stati utilizzati per verificare l'occupazione di spazio del kernel sia sul disco che in ram. Come è stato strutturato il filesystem e quanto spazio occupa. Quali sono i fattori che influenzano la latenza del boot, come misurare i tempi di boot, quali tecniche esistono e sono in sviluppo per ridurre i tempi impiegati dal boot. Si è descritto come e dove intervenire per ridurre l'occupazione di spazio sia da parte del kernel che da parte del filesystem. Nel quarto capitolo si è posta l'attenzione sull'importanza dei test, necessari a verificare la stabilità, la robustezza e la sicurezza del kernel linux.

Capitolo 1

Analisi del problema

La prima domanda che il progettista di sistemi embedded potrebbe porsi è relativa al perché usare Linux e il free software. Tecnicamente Linux è un kernel di classe Unix, creato da Linus Torvalds e oggi attivamente sviluppato da una vasta comunità di programmatori sparsa per il mondo. Generalmente il software che andrà a completare il sistema operativo (compilatori, linker, debugger, editor di testo, ambienti grafici, ecc) è parte del progetto GNU[1]. Alcune delle caratteristiche che il kernel Linux e il software libero offrono sono le seguenti:

- Completa configurabilità: è possibile configurare il kernel selezionando soltanto le features di cui si ha bisogno. É possibile compilare i driver per il solo hardware che si dovrà gestire, riducendo in tal modo lo spazio che sarà occupato dal kernel sia sulla memoria di massa che in ram.
- Aderente agli standard IEEE Posix: ciò garantisce che il software scritto per altri unix sia compilabile ed eseguibile su Linux apportando piccole o nessuna modifica al codice originale.
- Performance: molti sistemi embedded hanno bisogno di performance molto elevate. Se il sistema embedded ha la necessità di trattare grosse quantità di dati in tempi molto brevi, allora Linux è una scelta ottimale.
- Disponibilità del codice sorgente del kernel e di tutte le applicazioni che andranno a comporre il sistema operativo e gli strumenti di sviluppo. Gli sviluppatori e gli ingegneri hanno a disposizione una moltitudine di componenti già pronti, spesso sviluppati da diversi anni, ampiamente testati e supportati. La disponibilità di questo codice permette di concentrarsi solo sulle parti critiche che il sistema embedded dovrà gestire e controllare. Se il sistema embedded ha bisogno di una

1 Il fondatore del progetto GNU, acronimo ricorsivo di Gnu is Not Unix, e della FSF (Free Software Foundation) è Richard Stallman, www.gnu.org

particolare feature, in genere non è necessario aspettare mesi o anni prima che una software house la implementi, in quanto la disponibilità del codice sorgente rende possibile lo sviluppo della feature in modo autonomo.

- Scalabilità: Il kernel Linux è altamente scalabile. Supporta una moltitudine di processori e di architetture. É disponibile per i sistemi embedded, per i desktop e per i server. Può essere utilizzato su processori sia a 32 che a 64 bit, Intel e compatibili, su processori ARM, PowerPc, Motorola, Alpha, Sparc, Mips e su processori senza supporto alla MMU. Linux supporta una vastissima gamma di device. Il porting di un sistema embedded Linux si risolve spesso con una semplice ricompilazione per l'architettura target, raramente è necessario applicare delle patch al software per poter gestire delle condizioni particolari. Il grande vantaggio di GNU/Linux è che il software che gira sui sistemi embedded è lo stesso che si trova sui desktop o sui server. Ciò permette di utilizzare una macchina host, generalmente un normale pc, più performante e dotato di risorse hardware superiori rispetto alla macchina target, il sistema embedded, per sviluppare, testare e debuggare il sistema e le applicazioni che dovranno girare su di esso e quindi trasferire il sistema finito sul target. Sono disponibili alcuni tool che automatizzano la costruzione del filesystem root. Uno di questi, buildroot, permette di definire quale software dovrà essere aggiunto al filesystem root, scarica i sorgenti, applica le patch necessarie, crea la toolchain per una eventuale cross compilazione. L'utente ha la possibilità di aggiungere eventuali patch, aggiungere altri pacchetti software non previsti nella configurazione originale di buildroot semplicemente creando o modificando una serie di makefile.
- Libero da licenze proprietarie. La quasi totalità del software libero è rilasciato con licenza Gnu GPL[2] che garantisce all'utente quattro libertà. In breve, la licenza GPL garantisce la libertà di eseguire il programma per qualsiasi scopo (libertà 0), la libertà di esaminare il codice sorgente e di modificarlo secondo le proprie necessità

2 GNU General Public License, http://fsf.org

(libertà 1), la libertà di copiare il programma per aiutare il prossimo (libertà 2), la libertà di redistribuire copie modificate del programma in modo tale che tutti possano usufruire delle modifiche (libertà 3). Il codice coperto da licenza GPL, se modificato, deve essere rilasciato con la stessa licenza, in questo modo le eventuali migliorie apportate saranno disponibili a tutta la comunità. Un problema che potrebbe nascere è relativo al fatto che quando un software proprietario è linkato a codice coperto dalla GPL allora il software proprietario diventa un lavoro derivato di quello GPL e in quanto tale deve essere rilasciato con questa licenza. Per ovviare a questo problema e per permettere il link a librerie essenziali ad un sistema operativo, come le librerie C, è stata studiata la licenza LGPL, che oltre ai vantaggi offerti dalla GPL permette di implementare e di linkare alle librerie stesse del software che sarà rilasciato con licenza proprietaria. La libreria C glibc, che è necessaria per la corretta esecuzione di tutto il codice scritto in C, è rilasciata sotto la licenza LGPL e ciò è estremamente utile per software di cui non si vuole o non si può rilasciare il codice sorgente, come ad esempio software di tipo militare o crittografico o coperto da brevetti/licenze di terze parti. Una diretta conseguenza sia della GPL (LGPL) che dell'immensa disponibilità di codici sorgenti è l'inesistenza di costi legati alle licenze.

- Oltre alla glibc sono disponibili altre librerie C ottimizzate per i sistemi embedded e per processori senza MMU. Tra queste quelle che attualmente è la più avanzata è la libreria uClibc[3].
- Molte schede sono direttamente supportate da Linux, grazie ai produttori dell'hardware che hanno fornito ai mantenitori del kernel le patch e il codice necessario a gestirle in modo ottimale. Il crescente supporto ai sistemi embedded basati sul kernel Linux da parte di grandi aziende è stato appurato analizzando i dati ottenuti da un questionario[4] proposto a sviluppatori di sistemi embedded [12]. Dalle risposte ottenute si evince che molti programmatori sviluppano per Linux

3 http://www.uclibc.org
4 Munich/MIT Survey: The development of embedded Linux [12]

embedded non solo per esigenze lavorative ma anche nel loro tempo libero. Molto del codice prodotto è rilasciato con una licenza libera ed è specifico al particolare device costruito dal produttore di hardware. La percentuale di codice rilasciato è andata aumentando dal 2000 al 2004[5]. In base ai dati raccolti, gli autori sono giunti alla conclusione che le aziende produttrici di sistemi Linux embedded hanno condiviso il loro codice ricavandone vari benefici. Il codice rivelato è tipicamente generico, relativo al solo kernel. In pratica Linux è una commodity, è un prodotto che, semplicemente, offre una serie di servizi e di API per gestire al meglio una vasta gamma di device. Quello che valorizza e distingue un sistema embedded da un diretto concorrente sono le applicazioni che andranno a girare sul sistema operativo GNU/Linux.

Il kernel Linux è solo una parte, benché importante, del software che comporrà il sistema operativo. Per costruire ed avere un sistema embedded minimale è necessario tener conto anche del:

- Bootloader: responsabile del caricamento del kernel.
- File system root: il file system di base contenente le utility necessarie alla gestione e al funzionamento del sistema operativo. Per i sistemi embedded è necessario utilizzare dei filesystem atti a sfruttare le memorie Flash su cui il sistema andrà installato.
- Shell: l'interprete dei comandi, l'interfaccia tra l'utente o le applicazioni utente e il kernel.
- Compilatori, debugger e strumenti di sviluppo adeguati al processore e all'architettura su cui il sistema operativo andrà a girare.

[10], [11], [12], [37]

1.1 Requisiti hardware e software

Le specifiche dei requisiti hardware richiesti sono:

5 Al 17 maggio 2004, data di pubblicazione dei risultati del questionario

- Scheda di riferimento: Concurrent Technologies VP417, con architettura x86 dual core
- 512Mb di memoria ram
- Flash per alloggiare il Sistema Operativo. Il sistema operativo dovrà utilizzare uno spazio sulla flash inferiore a 50MB
- Periferiche da gestire: RS 485, 422, 232, Ethernet 10/100MB
- Hard Disk di appoggio IDE o SATA, da utilizzare per log
- ACPI[6], Watchdog, RTC[7], PXE per boot da remoto

La scheda Concurrent Technologies VP417 usa una architettura PC-AT progettata per applicazioni ad alte prestazioni e basata sul processore Intel Core 2 Duo. Il sistema operativo sviluppato non sarà alloggiato su nessun hard disk ma sarà installato sulla memoria flash (Application Flash), accessibile dal sistema operativo scelto (linux) utilizzando gli appositi driver MTD (Memory Technology Driver) forniti dal costruttore della scheda [8]. Della memoria, avente una dimensione di 64Mb è richiesto di utilizzarne una quantità non superiore a 50Mb. L'hardware da gestire comprende una porta ethernet e le classiche porte seriali. Si vuole utilizzare un hard disk di appoggio IDE o SATA per il log, ciò comporta l'inclusione nel kernel per la compilazione dei driver di entrambe le architetture. Il watchdog è un componente elettronico utilizzato per monitorare i possibili malfunzionamenti della scheda, nel qual caso si occupa di ripristinare il sistema resettando il processore. Il timer del watchdog viene impostato con un determinato valore, quindi inizia un conto alla rovescia. Se il timer non viene ciclicamente ripristinato al valore prestabilito, ma raggiunge lo zero, allora si presume che qualche componente non stia funzionando correttamente e il sistema è resettato. Alcuni timer watchdog svolgono dei compiti avanzati,monitorando la temperatura e il voltaggio della scheda.

I requisiti software relativi al sistema operativo sono i seguenti:

- Kernel 2.6.23 con patch Real Time rt13

6 Advanced Configuration and Power Interface
7 Real Time Clock

- Glibc, le librerie C devono essere compliant al compilatore gcc > 4.2.0
- Driver per le periferiche HW
- Protocolli UDP/IP e I2C[8]
- BusyBox
- Bash

Il kernel di riferimento è il 2.6.23 (kernel vanilla), al quale dovranno essere applicate le patch real time rt13. Nel kernel saranno compilati i driver e i protocolli per la gestione della rete. I2C è un bus sviluppato dalla Philips per essere utilizzato nelle Tv e che grazie alla sua semplicità d'uso è diventato velocemente uno standard. Il poco spazio occupato dal bus ne fa un elemento prezioso per i sistemi embedded. É utilizzato per lo scambio di piccole quantità di dati a bassa velocità e senza bisogno di grandi larghezze di banda tra i componenti IC dei sistemi embedded. I componenti che adottano I2C utilizzano solo due linee per i collegamenti, una utilizzata per il segnale di clock (SLC), l'altra per lo scambio seriale dei dati (SDA). Il dispositivo master e quello slave si sincronizzano tra di loro attraverso la linea del clock, e si scambiano i dati sulla linea SDA.

BusyBox è un programma estremamente utile per i sistemi embedded, utilizzato in sostituzione di una serie di programmi necessari alla gestione e all'amministrazione di un sistema linux. L'interprete dei comandi richiesto è bash

I servizi richiesti sul sistema operativo sono:

- NTP Client
- Ssh
- Gdb Server per remote debugging

Ntp (Network Time Protocol) client, come riportato nell'RFC 1305 fornisce un meccanismo atto a sincronizzare tra loro i client su reti di grandi dimensioni.

Ssh (Secure Shell) è un protocollo che permette di stabilire una sessione remota cifrata con un altro host. L'intera comunicazione (ovvero sia l'autenticazione che la sessione di

8 Inter-Integrated Circuit

lavoro) avviene in maniera cifrata. Per questo motivo, SSH è diventato uno standard di fatto per l'amministrazione remota di sistemi unix e di dispositivi di rete, rendendo obsoleto il protocollo telnet, giudicato troppo pericoloso per la sua mancanza di protezione contro le intercettazioni.

Il Gdbserver è utilizzato per effettuare il debug remoto di un programma utilizzando una linea seriale o attraverso una connessione TCP.

1.2 Sistemi operativi real time

I sistemi operativi e l'elaborazione real time sono fondamentali nella gestione di impianti di controllo di processo, nella robotica, nel controllo del traffico aereo, nelle telecomunicazioni, nei sistemi di comando e controllo militari. Lo standard POSIX 1003.1b definisce il real time come la capacità del sistema operativo di fornire i servizi richiesti in un intervallo di tempo ben definito. Quindi con il termine real time si indica una classe di sistemi operativi soggetti a limiti temporali che non possono essere in alcun modo superati. La correttezza di una elaborazione real time non è legata soltanto al risultato logico dell'elaborazione, ma è anche strettamente dipendente dal tempo in cui i risultati sono stati prodotti. Una elaborazione logicamente corretta ma fornita oltre il tempo massimo, la deadline, è considerata inaccettabile e il sistema ha fallito l'elaborazione. In generale in un sistema operativo real time solo alcuni dei processi o dei task sono a tempo reale ed hanno una priorità superiore agli altri processi del sistema. Ai processi real time, che devono reagire ad eventi che avvengono nel mondo esterno, si associa una scadenza che specifica un tempo di inizio o un tempo di completamento del compito. I processi real time si dividono in due categorie: hard real time e soft real time. I primi hanno scadenze temporali che non possono essere in nessun modo disattese, mentre i secondi hanno delle scadenze che è desiderabile che vengano rispettate. Consideriamo le differenze fra i tempi di risposta esistenti tra un normale processo, come ad esempio un editor di testo, un processo soft real time, qual è un player multimediale e un processo hard real time, come un programma che gestisce la sequenza di spegnimento del motore di

un razzo o un programma che gestisce un braccio robotico in una catena di montaggio. Se l'editor video non reagisce prontamente all'input dell'utente non effettuando un aggiornamento immediato del video o non acquisendo immediatamente il carattere battuto sulla tastiera, solo pochi utenti potrebbero accorgersi del ritardo. Se il player multimediale, durante la riproduzione di un filmato o di una canzone, non riesce a decodificare alcuni frame allora l'utente potrebbe avere un effetto sgradevole dalla riproduzione, ma la fruizione dell'opera non subirebbe ripercussioni eccessivamente negative. Nel caso dello spegnimento del motore del razzo la sequenza deve essere esattamente rispettata, altrimenti il motore potrebbe esplodere o il razzo potrebbe finire fuori traiettoria. Analogamente il braccio robotico deve rispettare le scadenze temporali a cui è soggetto, altrimenti potrebbe ad esempio fallire la saldatura dei componenti che passano sulla catena di montaggio, non saldando alcune parti e saldandone, senza bisogno, altre.

I sistemi operativi real time devono possedere le seguenti caratteristiche: [6]

- Multitasking/multithreading: i sistemi operativi real time devono supportare il multitasking e il multithreading
- Priorità: Ad ogni task deve essere associata una priorità, i task che devono eseguire compiti critici devono avere priorità alte.
- Ereditarietà della priorità: questi sistemi operativi devono avere un sistema che supporti l'ereditarietà delle priorità.
- Prelazione: I sistemi operativi real time devono essere prelazionabili. Un processo/ task ad alta priorità, pronto ad essere eseguito, deve poter sempre prelazionare un processo/task a priorità più bassa.
- Sincronizzazione e comunicazione tra processi: Comunemente, nei sistemi embedded, la comunicazione tra task avviene attraverso lo scambio di messaggi. Nei RTOS lo scambio dei messaggi dovrebbe avvenire con tempi costanti. La sincronizzazione tra task dovrebbe avvenire attraverso l'uso di mutex e semafori.
- Allocazione di memoria dinamica: L'allocazione della memoria dovrebbe avvenire con tempi certi

- Latenza dell'interrupt: è il tempo che intercorre tra l'istante in cui un segnale di interrupt viene ricevuto all'istante in cui viene chiamata la corrispondente routine. I sistemi operativi real time devono avere una latenza di interrupt che sia predicibile e la più breve possibile.
- Latenza di scheduler: è il tempo che intercorre tra l'istante in cui un task diventa pronto all'esecuzione e l'istante in cui viene mandato in esecuzione. Come per la latenza di interrupt, anche la latenza di scheduler deve essere deterministica.

I sistemi operativi real time possono essere caratterizzati dai seguenti requisiti: [5]

- Determinismo
- Prontezza
- Controllo utente
- Affidabilità
- Operatività fail-soft

Un sistema operativo è deterministico se effettua le operazioni a tempi fissati e predeterminati o in intervalli di tempo predeterminati. Nei sistemi operativi real time le richieste di servizio dipendono da eventi esterni. Il limite temporale entro cui il SO soddisfa le richieste dipende in primo luogo dalla velocità con cui può rispondere alle interruzioni e, in secondo luogo, dalla capacità del sistema di gestire le richieste entro il tempo richiesto. Per misurare quanto un sistema operativo può operare deterministicamente si usa il ritardo massimo o latenza, da quando arriva un'interruzione da un dispositivo a quando l'interruzione viene servita. Nei sistemi operativi real time il ritardo massimo deve essere dell'ordine di pochi microsecondi.

La prontezza è collegata al determinismo. Il determinismo riguarda il ritardo del sistema operativo prima di riconoscere un'interruzione, la prontezza è relativa invece, al tempo impiegato dal sistema operativo a servire l'interruzione dopo che questa è stata riconosciuta. La prontezza, il tempo impiegato a servire l'interruzione, è soggetto a diversi fattori. La quantità di tempo necessaria ad iniziare la routine di servizio dell'interruzione può variare in funzione del fatto che sia necessario un cambio di processo o meno. Se

bisogna cambiare contesto, allora il ritardo sarà più lungo rispetto a quello che si avrà se l'ISR è servita nel contesto del processo corrente. Il tempo dipende poi dall'hardware su cui sta eseguendo l'interrupt. Infine il ritardo può essere influenzato da interruzioni annidate, se il servizio di una interruzione è bloccato a causa di un altra ISR, il ritardo inevitabilmente sarà superiore.

Determinismo e prontezza formano il tempo di risposta agli eventi esterni, uno dei parametri critici per i sistemi real time.

Con il controllo utente i sistemi operativi real time permettono all'utente di modificare le priorità dei singoli task, modulando la schedulazione dei processi secondo le proprie necessità.

L'affidabilità è un altro aspetto fondamentale dei sistemi real time. Vista la delicatezza di molti sistemi che operano in tempo reale, le prestazioni devono essere garantite, non possono degradare.

Con operatività fail-soft ci si riferisce alla capacità del sistema di preservare la maggior quantità possibile di dati in caso di fallimento. Generalmente i sistemi real time, in caso di malfunzionamento, cercano o di correggere il problema o di minimizzare gli effetti del guasto, continuando in ogni caso l'esecuzione. Un aspetto importante dell'operatività fail-soft è la stabilità. Il sistema è stabile se in caso di malfunzionamento riesce a garantire le scadenze dei task con priorità più alta.

Come detto uno dei parametri temporali più importanti per i sistemi real time è la latenza o

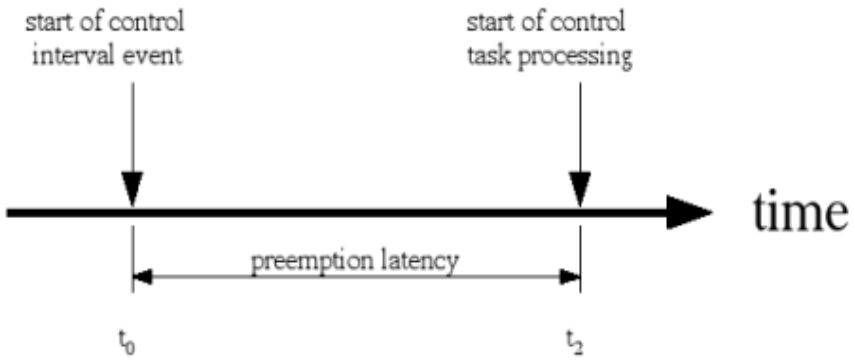


Figura 1: Preemption Latency [15]

ritardo massimo, cioè il tempo che intercorre tra l'istante in cui avviene l'evento che innesca la gestione dell'interruzione e l'istante in cui l'interruzione è effettivamente servita (figura 1). La latenza è ottenuta come somma di ritardi più brevi [15]: un tempo dipendente dalla risposta hardware, un tempo relativo al servizio dell'interrupt, un tempo dovuto al contex switch. La latenza può essere rappresentata nel modo seguente (figura 2):

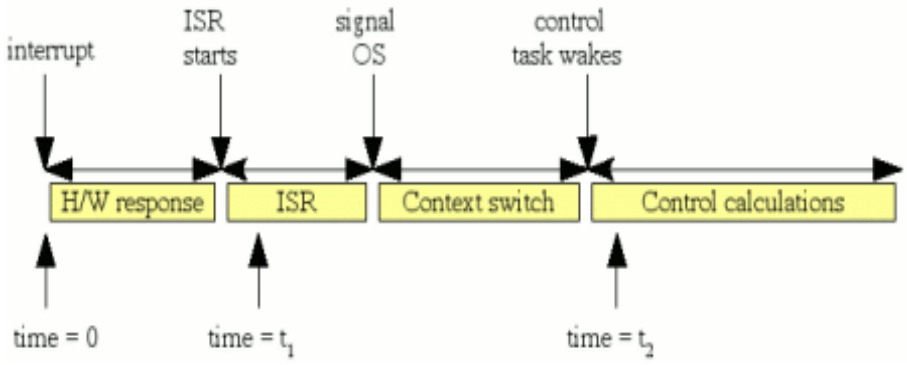


Figura 2: Components of Latency [15]

All'istante di tempo t=0 si genera l'interrupt

All'istante t=t1 inizia la gestione dell'interrupt con l'invocazione della opportuna routine, il tempo t1 è il ritardo dovuto all'interrupt

All'istante t=t2, il sistema operativo ha effettuato il contex switch ed il processo viene mandato in esecuzione.

Il ritardo dovuto alla gestione dell'interrupt è uno dei principali motivi di non determinismo nei tempi di risposta. Si possono verificare alti tempi di latenza nella gestione delle interrupt a causa della disabilitazione delle interruzioni per un lungo periodo di tempo o a causa di una errata registrazione nel kernel dell'handler che gestirà l'interruzione. Se il kernel o un driver devono eseguire del codice che non è possibile interrompere allora è necessario proteggerlo disabilitando il sistema delle interruzioni. Ciò permette l'esecuzione di codice critico, ma ha come controparte l'innalzamento della latenza. Le interruzioni sono generalmente gestite da device driver, i quali registrano presso il kernel un handler al codice che verrà eseguito in caso di interruzione. L'handler può essere registrato sia come fast interrupt che come slow interrupt. Le interruzioni sono

automaticamente disabilitate nel caso in cui si stia eseguendo un fast interrupt, mentre rimangono abilitate se si esegue uno slow interrupt. Nel primo caso il dispositivo ad alta priorità potrà eseguire l'handler dell'interrupt senza essere interrotto da una nuova eventuale interruzione. Nel secondo un dispositivo ad alta priorità potrà sempre interrompere il dispositivo a priorità inferiore. Se il dispositivo a bassa priorità registra il suo driver come fast interrupt o il dispositivo ad alta priorità registra il proprio driver come slow interrupt si osserverà un innalzamento della latenza totale. La durata dell'interrupt handler è sotto il controllo dello scrittore del codice relativo all'ISR (interrupt service routine), tuttavia se la ISR ha una componente softirq allora si può introdurre un ritardo non deterministico [6]. Affinché si abbia una bassa latenza di interrupt è necessario che la ISR compia pochi compiti, come il settaggio di alcuni registri, e lasci il grosso del lavoro, come ad esempio l'elaborazione dei dati, a routine eseguite al di fuori dell'interrupt handler. In questo modo l'interrupt handler può essere diviso in due parti, la prima metà che si occupa del settaggio dei registri e la softirq, la seconda, che si occupa della rimanente elaborazione. La softirq viene eseguita con il sistema delle interruzioni abilitato ed è la parte di codice soggetta a prelazione, sezioni critiche escluse, nel caso in cui durante la sua elaborazione avvenga una nuova interruzione da parte di un task con priorità superiore a quella del task attuale. Ciò implica che per evitare ulteriori ritardi, le ISR dei dispositivi real time non devono avere nessuna softirq, e tutto il lavoro deve essere svolto nella prima metà.

Oltre al ritardo introdotto dalla gestione delle interruzioni un altro contributo alla latenza, che in realtà è il principale contributo ai ritardi del kernel, è dato dalla latenza introdotta dallo scheduler. Lo scheduler di Linux, fino alla versione 2.4, introduceva dei ritardi a causa del fatto di essere senza prelazione. La schedulazione del prossimo processo, negli scheduler a time sharing avviene al ritorno da una interruzione o al ritorno da una chiamata di sistema. Se un processo a bassa priorità sta eseguendo una chiamata a sistema e si trova in kernel mode, un processo a priorità più alta non potrà prelazionarlo fintanto che il processo in esecuzione non sarà tornato in user mode. La natura non prelazionabile

del kernel introduce dei ritardi che variano in funzione del tipo di chiamata di sistema che si sta servendo e vanno da alcune decine a centinaia di millisecondi. La disabilitazione delle interruzioni, indirettamente contribuisce alla latenza introdotta dallo scheduler. Uno degli instanti in cui lo scheduler seleziona il prossimo processo è in relazione alle interruzioni del timer. Se le interruzioni sono disabilitate allora anche lo scheduler salterà alcuni istanti di possibile prelazione, incrementando la latenza. Il ritardo introdotto dallo scheduler è dato dalla somma del tempo necessario a selezionare il prossimo processo e del tempo necessario ad effettuare il contex switch. Originariamente il kernel Linux è stato studiato per i server e per i desktop. La durata dello scheduler cresce linearmente in funzione del numero di processi pronti presenti nel sistema. Tutti i processi pronti ad essere eseguiti, compresi quelli real time, sono contenuti nella stessa coda e ogni volta che lo scheduler deve scegliere quale tra questi processi mandare in esecuzione deve scorrere la lista alla ricerca del processo a priorità più alta. Maggiore è il numero di processi pronti contenuti nella lista, maggiore sarà il tempo impiegato dallo scheduler per compiere la selezione. Le versioni standard del kernel Linux sono studiate per la condivisione delle risorse tra i processi, l'algoritmo di scheduling utilizzato di default è time sharing e implementa la politica di scheduling SCHED_OTHER. Per i processi che necessitano di una maggiore interattività sono disponibili algoritmi di scheduling che implementano le politiche SCHED_FIFO e SCHED_RR. Tuttavia queste implementazioni sono adatte solo a processi soft real time e non garantiscono i processi hard real time. Quando si parla di real time e di Linux (kernel standard) si intende implicitamente soft real time. Negli anni sono state sviluppate alcune soluzioni per rendere hard real time il kernel Linux. Sono state sviluppate delle patch da MontaVista e Red Hat per ridurre la latenza del kernel e migliorare le prestazioni. Tra le soluzioni avanzate se ne esamineranno alcune, si vedrà come RTAI e RTLinux hanno trasformato Linux in un kernel hard real time e come, lo stesso risultato sia stato raggiunto grazie alle patch di MontaVista. Grazie a queste soluzioni Linux è ascrivibile tra i sistemi operativi hard real time.

Per soddisfare i requisiti dei sistemi real time, i sistemi operativi devono avere un context

switch il più veloce possibile, devono rispondere velocemente alle interruzioni, devono essere di piccole dimensioni, gestire il multitasking e fornire degli strumenti di comunicazione tra i processi, devono avere uno schedulatore con prerilascio basato sulle priorità, devono minimizzare gli intervalli di tempo in cui le interruzioni sono disabilitate, devono poter ritardare, mettere in pausa e far ripartire i task e devono avere speciali timeout. L'elemento più importante dei sistemi operativi real time è lo scheduler a breve termine, il quale non deve avere come obiettivo l'equità o la minimizzazione del tempo medio di risposta, bensì deve garantire che i processi hard real time possano essere mandati in esecuzioni entro tempi certi e fissati.

1.3 La soluzione Dual Kernel

Le proposte che finora hanno riscosso i maggiori consensi usano una struttura a doppio kernel (figura 3). Il kernel Linux è trattato come un processo avente priorità più bassa rispetto ai processi real time. In queste condizioni Linux può essere eseguito solo quando non ci sono processi real time da mandare in esecuzione. Il kernel Linux può essere prelazionato e non può disabilitare le interruzioni, che sono gestite dal kernel real time e sono inviate al kernel Linux solo se non c'è nessun processo real time che le intercetta. Se Linux disabilita il sistema delle interruzioni, il kernel real time semplicemente evita di notificare l'eventuale interruzione a Linux. In queste condizioni Linux si occupa di gestire le sole attività che non sono real time, tra cui la gestione dei processi utente a bassa priorità. Le soluzioni hard real time per Linux sono sostanzialmente RTLinux e RTAI.

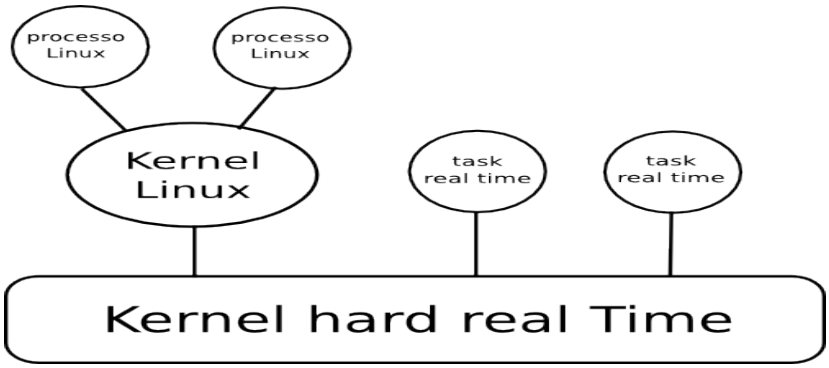


Figura 3: Sistema dual kernel

Entrambe adottano il doppio kernel, entrambe intercettano le interruzioni con il kernel real time, che le passerà, in seguito, al kernel Linux se non sono attivi task real time.

1.3.1 RTLinux, RealTime Linux

RTLinux [13] [14] è un kernel hard real-time sviluppato nell'Istitute of Tecnology del New Mexico nel 1994 da Michael Barabanov con la supervisione dal professor Victor Yodaiken. Con la soluzione adottata da Barabanov e Yodaiken, il sistema operativo real time e il sistema linux a time-sharing cooperano. Linux è trattato come un processo background del kernel real time, schedulato quando nessun processo real time richiede di essere eseguito. Linux non può accedere alle interruzioni disabilitandole e non può evitare di essere prelazionato. Il sistema che permette tutto ciò è rappresentato da una emulazione software del sistema delle interruzioni. Se Linux tenta di disabilitare le interruzioni, il kernel real time intercetta la richiesta e fa credere a Linux che sia stata soddisfatta, in questo modo il kernel real time potrà gestire in modo ottimale tutte le interruzioni senza che Linux inserisca dei ritardi. Se l'interrupt in arrivo è gestito da un processo real time, il controllo verrà passato a questo processo. Se l'interruzione non è gestita da nessun processo real time o se si desidera condividere con Linux la gestione dell'interruzione, allora l'interrupt è marcata “in attesa”. Se Linux non ha disabilitato il sistema delle interruzioni allora lo strato di emulazione gli passa l'interrupt che verrà gestito in modo classico. Il principale vantaggio dell'uso dell'emulazione è relativo al fatto che il kernel real time non è in alcun modo influenzato dallo stato di Linux. Se Linux è in kernel mode o in user mode, se sta disabilitando o abilitando le interruzioni, il kernel real time è in grado di rispondere alle interruzioni con una latenza minima. Il kernel real time non deve mai attendere che Linux liberi delle risorse. RTLinux comunica con Linux e con i processi non real time sia attraverso l'uso di una memoria condivisa, che attraverso delle apposite entry create nella directory /dev. RTLinux deve essere predicibile, semplice, veloce e con un sovraccarico minimo. A Linux sono lasciate la gestione dell'inizializzazione delle periferiche, del sistema e delle risorse non condivisibili. RTLinux è un sistema modulare,

viene caricato da Linux come insieme di moduli. Il core di RTLinux è un componente che permette di installare degli interrupt handlers aventi latenza molto bassa e delle routine a basso livello per gestire la sincronizzazione e le interruzioni. Il core è poi esteso utilizzando una serie di moduli caricabili dal kernel. Questi moduli comprendono uno scheduler, un modulo per la memoria condivisa, uno per controllare i timer e per le comunicazioni I/O. Due dei principali moduli che compongono RTLinux sono lo scheduler e il modulo che implementa le FIFO RT. Se si desiderasse utilizzare uno scheduler diverso da FIFO RT, sarebbe sufficiente caricare un nuovo modulo. Lo scheduler standard di RTLinux manda in esecuzione il processo che, fra quelli pronti ha la priorità maggiore, e questo rimane in esecuzione finché non rilascia la cpu o un processo a priorità più alta non lo prelaziona.

1.3.2 RTAI, Real Time Application Interface

Come RTLinux anche RTAI, Real Time Application Interface, è un kernel hard real time basato su una struttura a doppio kernel. RTAI è stato creato al Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano dal professor Paolo Mantegazza. Come RTLinux, RTAI è formato da una serie di moduli del kernel caricati a run time. Il nucleo di RTAI è l'Hardware Abstraction Layer (HAL), al di sopra del quale girano sia il kernel linux che i processi hard real time di RTAI (figura 4). Lo scopo fondamentale di HAL è quello di separare le applicazioni hard real time e quelle time sharing, assegnando ai processi hard real time una priorità più alta rispetto a quella che viene assegnata al kernel Linux e alle applicazioni che girano su di esso. Hal permette quindi di minimizzare le latenze dei processi hard real time e di far girare le applicazioni time sharing e il kernel Linux negli istanti di inattività dei processi a tempo reale. Le interruzioni provenienti dall'hardware sono intercettate dall'HAL, che le smista a Linux soltanto se nessun processo hard real time ne ha richiesto la gestione, nel qual caso l'interruzione è inviata dall'HAL direttamente al processo. Sfruttando l'hardware abstraction layer il kernel RTAI riesce ad ottenere il pieno controllo sul sistema delle interruzioni e ad effettuare la prelazione sia dei

processi che del kernel Linux. I processi hard real time sono creati e schedulati utilizzando le API fornite da RTAI. Per schedulare i processi real time RTAI usa il proprio scheduler. Attraverso la IPC fornita da RTAI, i processi real time possono comunicare con i processi time sharing e con il kernel Linux, permettendo di fatto ai processi di RTAI di effettuare delle chiamate di sistema al kernel.

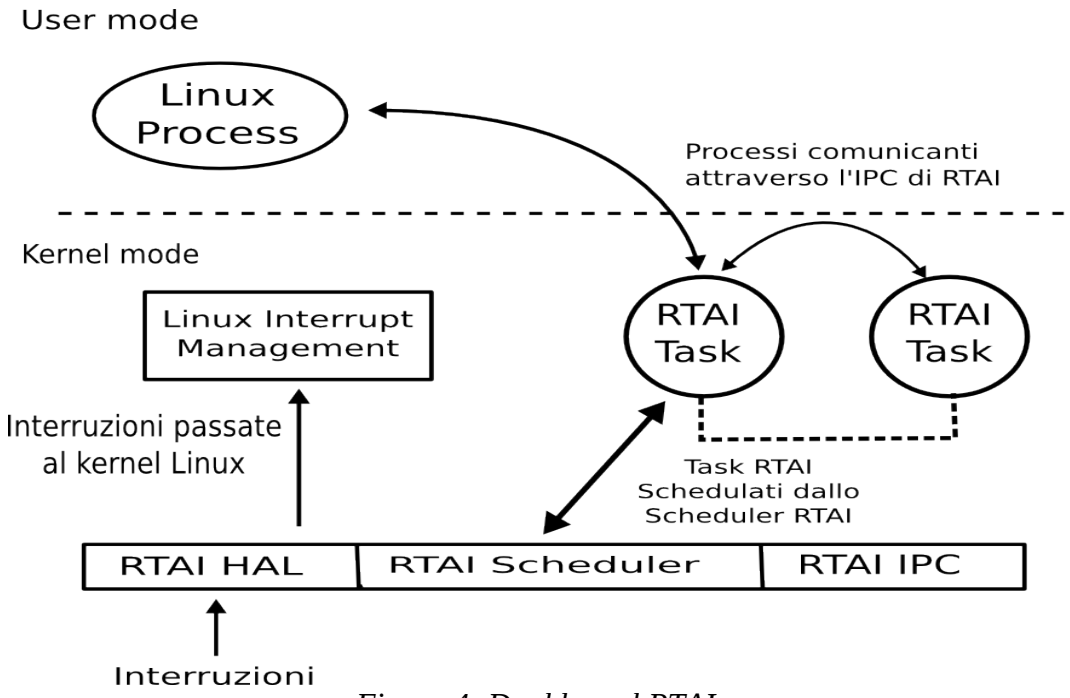


Figura 4: Dual kernel RTAI

Tuttavia è opportuno limitare queste chiamate alla fase di inizializzazione e terminazione del processo real time. Una chiamata di sistema al kernel Linux da parte di un processo real time potrebbe di fatto introdurre delle latenze. [6], [16], [52]

1.4 La soluzione di MontaVista.

La soluzione real time proposta da MontaVista è basata sulla Preemption patch, sulla Low-Latency patch e sullo scheduler real time O(1).

1.4.1 Preemption patch

La preemption patch nasce dall'osservazione che era possibile effettuare, in modo sicuro, la prelazione di processi in esecuzione in kernel mode, se questi non stavano eseguendo

nessuna sezione critica protetta da spin lock[9] [6]. La patch, introdotta durante lo sviluppo di quello che sarebbe diventato il ramo stabile del kernel 2.6 è attualmente mantenuta da Robert Love. La preemption patch rende il kernel prelazionabile, permettendo la sospensione del task che in un determinato momento è in kernel mode se un altro task con priorità superiore è pronto per essere eseguito. É stato osservato che grazie alla patch i tempi medi di latenza subiscono un netto miglioramento, riducendosi a circa 1 ms e migliorando la responsitività complessiva del sistema [17]. Per poter effettuare la prelazione del kernel, è stata modificata la struttura che descrive il processo, introducendo il membro *preempt_count*. Se preempt_count è zero allora il kernel può essere prelazionato senza rischi, se la variabile ha un valore diverso da zero allora la prelazione non è possibile. Per operare su preempt_count sono state introdotte due macro: *preempt_disable* e *preempt_enable*, la prima disabilita la prelazione incrementando preempt_count, la seconda decrementa il valore di preempt_count e quando questa raggiunge lo zero allora la prelazione è abilitata. Le routine di spinlock sono state modificate in modo tale da chiamare la macro preempt_disable in entrata e chiamare preempt_enable in uscita. Analogamente è stato modificato il codice assembly relativo al ritorno da una interrupt o da una chiamata di sistema per esaminare il contenuto di preempt_count prima di decidere se effettuare o meno una nuova schedulazione.

1.4.2 Low Latency patch

La Low-Latency patch [6], scritta da Ingo Molnar e attualmente mantenuta da Andrew Morton, affronta il problema della riduzione della latenza dello scheduler inserendo dei punti di schedulazione all'interno dei blocchi di codice del kernel eseguiti per lungo tempo. Periodicamente, in corrispondenza dei punti di schedulazione, il kernel verifica se un qualche processo a priorità superiore di quello che è attualmente running ha richiesto di essere schedulato. I blocchi di codice identificati per l'inserimento dei punti di

9 Gli spinlock sono utilizzati per realizzare la mutua esclusione attraverso un lock busy-wait. Quando il lock è libero, un thread può occuparlo, operare in modo esclusivo, e quindi rilasciato. Se il lock non è libero, il thread deve attendere finché non diventa disponibile.

schedulazione sono i cicli su grosse strutture dati. Il task che ha bisogno di essere schedulato setta la variabile need_resched. Per poter identificare in quali blocchi di codice poter inserire i punti di schedulazione, Morton ha scritto una apposita patch, la rtc-debug patch, per il driver del clock real time. Quando la latenza dello scheduler supera una determinata soglia, viene scritto sul file del log di sistema il contenuto dello stack, in modo tale da poter identificare la routine che ha causato la latenza. Questa patch fornisce i risultati migliori se utilizzata congiuntamente alla preemption patch. [6], [33], [50]

1.4.3 Scheduler O(1)

Il numero di cicli impiegati dallo scheduler per stabilire quali task mandare in esecuzione, per quanto a lungo e su quale CPU, in caso di macchine multiprocessore, insieme al tempo impiegato a compiere il contex switch sono direttamente proporzionali al numero di processi presenti nel sistema. Maggiore è il numero dei processi, maggiori saranno le latenze introdotte dallo scheduler. Il classico scheduler O(n) è stato sostituito dallo scheduler O(1) sviluppato da Ingo Molnar. Lo scheduler O(1) impiega sempre un lasso di tempo costante sia per schedulare il prossimo processo sia per effettuare il contex switch, rendendosi di fatto indipendente dal carico del sistema. L'algoritmo implementa due code, una per i *processi attivi* e l'altra per i *processi expired*.

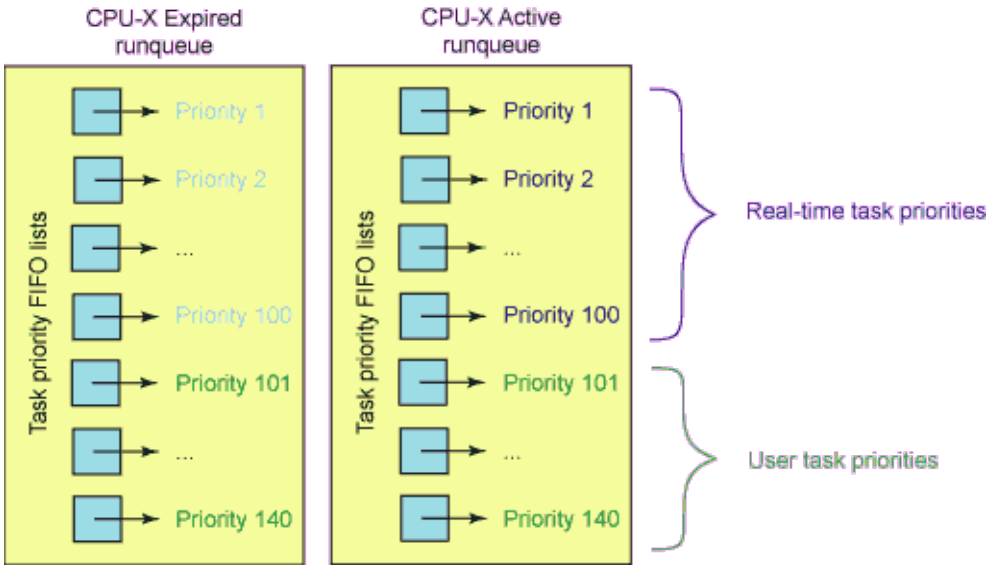


Figura 5: Struttura delle code dello Scheduler O(1)

I processi real time condividono tutti la stessa coda. Entrambe le code sono ordinate in base alla priorità, per ognuna delle quali è mantenuta una lista dei processi pronti ad essere eseguiti. Gli indici delle code sono mantenuti in una bitmap, in questo modo la ricerca del processo a priorità più alta si riduce ad una ricerca O(1). Quando un processo ha esaurito il proprio quanto di tempo è spostato nel vettore dei processi expired e contemporaneamente il suo quanto è ripristinato. Nel momento in cui la coda dei processi attivi è vuota, lo scheduler inverte le due code, facendo diventare attiva la coda expired e viceversa, quindi procede alla schedulazione del processo a priorità più alta. Tutte queste operazioni sono molto veloci essendo compiute su puntatori. Lo scheduler O(1) ha il pregio di operare sempre su code ordinate in base alla priorità dei processi, in questo modo non sarà necessario scorrere liste composte da n processi alla ricerca di quello a priorità maggiore da mandare in esecuzione. Lo scheduler O(1) elimina di fatto il *goodness loop* e il *recalculation loop*. Il goodness loop indica il ciclo che lo scheduler deve effettuare per scorrere la lista dei processi pronti alla ricerca del task a priorità più alta da mandare in esecuzione. Con recalculation loop si indica invece il ciclo da effettuare sui processi che hanno esaurito il loro quanto di tempo ed hanno bisogno che venga ricalcolato. Entrambi questi cicli dipendono dal numero di processi (real time o meno) presenti nel sistema. Lo schedulatore, infine, riconosce 140 livelli di priorità, di questi i primi 100 sono occupati dai processi real time i rimanenti 40 sono dedicati ai processi time sharing (figura 5).

Per poter definire un task real time in linux bisogna considerare tre parametri:

- Scheduling Class
- Priorità del processo
- Timeslice

Scheduling Class; Linux offre tre classi di schedulatore, due relativi a processi real time, uno per processi non real time. Le tre classi sono:

- SCHED_FIFO: Implementa la politica First In First Out per i processi real time. Un processo gestito con questa politica rimarrà running fintanto che non si bloccherà su di una operazione di I/O, non sarà prelazionato da un processo a

priorità più alta o non rilascerà volontariamente la CPU. Se il processo è prelazionato da un altro processo di priorità più alta, sarà inserito in testa alla lista dei processi aventi la sua stessa priorità, in modo tale da essere schedulato per primo quando il processo che lo ha prelazionato terminerà la sua esecuzione. Alla fine dell'esecuzione il processo è inserito in coda alla lista dei processi aventi priorità identica alla sua.

- SCHED_RR: La politica implementata dallo schedulatore real time Round Robin è simile a quella implementata dallo scheduler SCHED_FIFO, con la differenza che i processi possono essere running solo per un quanto di tempo. Terminato il quanto il processo è prelazionato ed è inserito in coda alla lista dei processi con la sua priorità. Se il processo è prelazionato da un processo con priorità più alta, sarà schedulato di nuovo per poter terminare il quanto di tempo, non appena il processo che lo ha prelazionato termina
- SCHED_OTHER: In questa classe si colloca il classico schedulatore time-sharing di linux, utilizzato per i processi non real time.

Priorità; I processi non real time, gestiti con la politica time-sharing, hanno una priorità pari a zero. I processi real time, sia che siano schedulati con SCHED_FIFO che con SCHED_RR, hanno una priorità che varia in un range tra 1 e 99. Maggiore è il numero assegnato, maggiore è la priorità. I processi real time hanno quindi sempre una priorità maggiore di quella dei processi time sharing. La priorità dei processi real time può essere manipolata tramite le funzioni *sched_getparam* e *sched_setparam*, la prima setta la priorità, la seconda la legge. Queste funzioni sono ininfluenti se chiamate su processi time sharing.

TimeSlice: É l'intervallo di tempo assegnato al processo running prima che venga prelazionato. Questo valore ha effetto solo se si usa la politica SCHED_RR. Un processo gestito con politica SCHED_FIFO, teoricamente, potrebbe rimanere running all'infinito, finché non decide di rilasciare la CPU. [6], [18], [19], [50], [51], [53]

1.5 MTD e Journaling Flash File Systems

Nei sistemi embedded è consuetudine utilizzare memorie a stato solido come memorie di massa. La caratteristica principale delle memorie a stato solido è legata al fatto di essere non volatili, i dati scritti su di esse rimangono persistenti anche dopo la rimozione dell'alimentazione elettrica. Sono costruite con due diverse tecnologie: NOR e NAND. Molti sistemi embedded utilizzano le memorie flash solo per caricare il sistema operativo che dovranno eseguire, altri le utilizzano come dei veri e propri hard disk. In Linux questi dispositivi sono gestiti tramite il sottosistema MTD (Memory Technology Device).

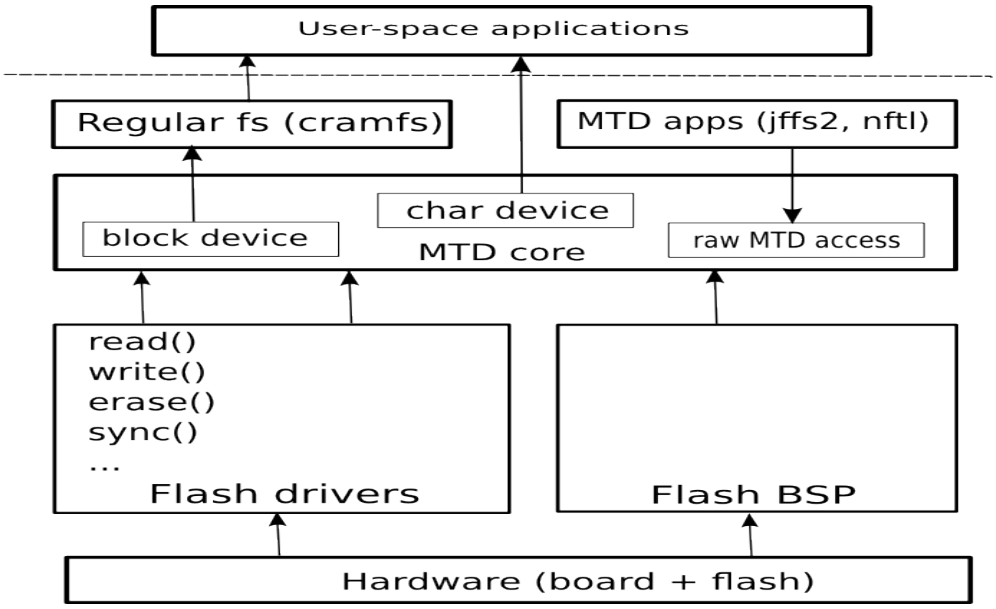


Figura 6: Architettura MTD [6]

Tradizionalmente per accedere alle memorie flash si utilizzava il Flash Translation Layer (FTL) che emulava i device a blocchi e permetteva la creazione di file system sulla flash. Per superare i limiti di FTL è stato sviluppato il sottosistema MTD, che è una combinazione di driver a basso livello con una interfaccia ad alto livello. Il sottosistema MTD non implementa nuovi driver, ma mappa le memorie flash sui driver per i dispositivi a blocchi e a caratteri. Il driver che gestisce la flash registra presso il sottosistema MTD una serie di callback che saranno richiamate per realizzare le operazioni di I/O (figura 6).

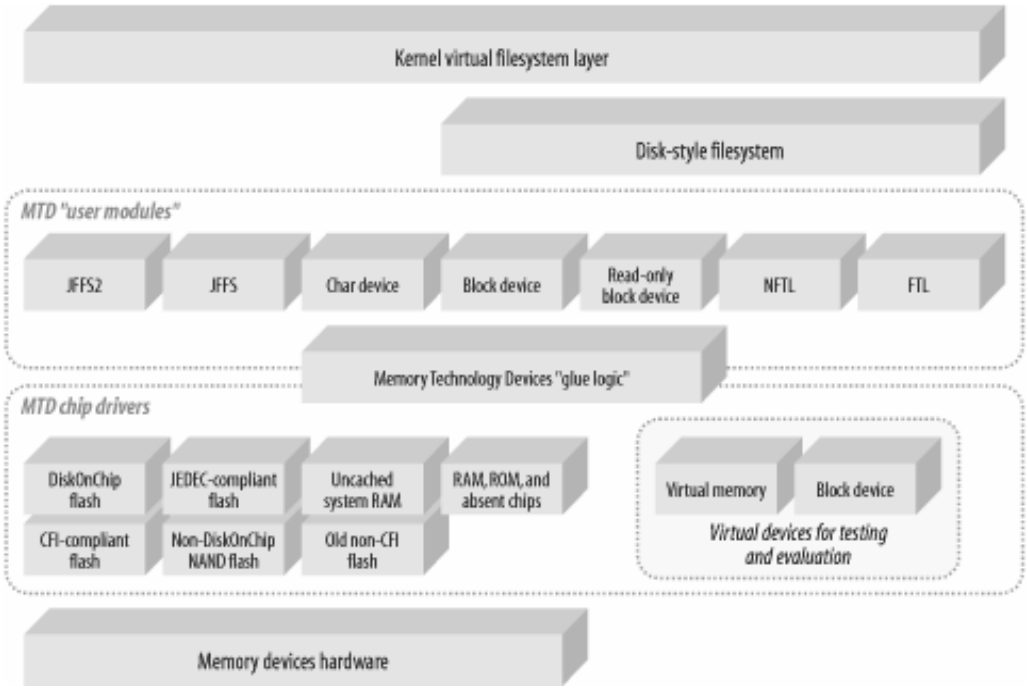


Figura 7: The MTD subsystem [7]

L'architettura MTD è composta da:

- Core MTD: implementa i device a blocchi e a carattere, rappresenta l'interfaccia tra i driver a basso livello della flash e lo strato applicativo
- Driver flash a basso livello: i driver relativi ai chip NOR o NAND
- Flash BSP: Il layer BSP permette al driver della flash di lavorare direttamente con il processore e con la scheda su cui è montata.
- MTD Applications: comprende moduli del kernel, come il filesystem jffs2, o applicativi user-space.

Le flash sono fortemente dipendenti dalla tecnologia utilizzata per la loro costruzione. Le flash basate su tecnologia NOR sono le più vecchie e offrono le migliori prestazioni in fase di lettura a discapito della capacità. Le flash NAND sono più recenti, offrono alte capacità, alte velocità sia in fase di lettura che di scrittura/cancellazione dei dati, tutto ciò al prezzo di una interfaccia input/output piuttosto complicata. Le flash possono essere gestite come dei veri e propri hard disk, possono essere partizionate e divise in blocchi. L'operazione di scrittura sulle memorie flash avviene cambiando i bit che si vogliono

scrivere dal valore 1 al valore 0, tuttavia per poter scrivere/cancellare anche un solo bit è necessario cancellare l'intero blocco all'interno del quale i bit modificati risiedono. Ciò implica che tutti i dati validi presenti in quel blocco debbano essere spostati prima della scrittura. A differenza delle NOR, i blocchi delle NAND sono normalmente divisi in pagine di 512 bit e ad ogni pagina sono poi associati altri 16 bit di spazio extra utilizzati per conservare i metadati e i codici di correzione degli errori. La necessità di manipolare interi blocchi per poter cancellare anche un solo bit dipende dalle caratteristiche costruttive delle memorie e in ultima analisi rappresenta il principale problema di questi dispositivi. Sulle flash NOR, per ogni blocco, è possibile effettuare circa 100.000 cicli di cancellazione, mentre sulle flash NAND i blocchi possono essere cancellati circa un milione di volte. Il ciclo di vita di questi dispositivi è quindi legato al numero massimo di operazioni di cancellazione che è possibile effettuare. I file system tradizionali (ext2/ext3) non sono adatti per i sistemi embedded e per le memorie a stato solido, per poter sfruttare efficacemente le flash ne sono stati sviluppati di nuovi. Attualmente i filesystem più utilizzati sono il CRAMFS, lo SquashFS, il JFFS[10] e la sua evoluzione JFFS2.

Se il sistema embedded ha un file system che non deve essere modificato nel tempo, allora si può scegliere di utilizzare uno tra Cramfs e Squashfs. Il Cramfs è stato sviluppato da Linus Torvalds. É un filesystem compresso e a sola lettura, particolarmente adatto per quei sistemi in cui si vuole privilegiare la stabilità e la sicurezza, coniuga semplicità e ottimizzazione dello spazio. Per comprimere i dati usa la libreria zlib mentre i metadati associati al file system rimangono non compressi. Una volta creata l'immagine del filesystem, questa è trasferita sulla flash utilizzando l'utility mkcramfs. Anche SquashFs è un filesystem compresso e a sola lettura, diffusamente utilizzato per la creazione di live cd. La compressione è ottenuta oltre che con le zlib anche con il Lembel-Ziv-Markov Algorithm (LZMA).

Se il filesystem non è a sola lettura, allora la scelta ricade su JFFS o JFFS2. Sono entrambi file system log-structured, il filesystem JFFS2 è semplicemente una lista di nodi o log,

10 Journaling Flash File System

ogni modifica al file è registrata in un log, che è poi salvato direttamente sulla flash. Il log contiene una serie di informazioni relativamente ai dati da salvare, alla loro dimensione, ai metadati associati ai dati, e una serie di indici per risalire al file a cui il nodo appartiene e all'offset dei dati all'interno del file. Le informazioni ausiliarie registrate nel nodo sono necessarie per poter ricostruire il file in fase di lettura. JFFS risolve alcuni problemi legati alla natura dei dispositivi a stato solido, tra i quali: il *garbage collection*, la *gestione dei blocchi difettosi* e il *wear leveling*[11]. Con garbage collection si identifica il processo di recupero dei blocchi che contengono dei dati marcati come non validi. Per poter recuperare questi blocchi è necessario spostare i dati validi in un nuovo blocco e cancellare quello vecchio rendendolo di nuovo disponibile. Con gestione dei blocchi difettosi si intende invece il processo di identificazione di quei blocchi, che a causa di usura o per un difetto di fabbricazione non sono utilizzabili. Quando un blocco è inutilizzabile, viene marcato come tale ed è indicizzato in una apposita tabella. Alcuni costruttori implementano questa funzione in hardware, in un microcontroller associato alla flash. Infine con wear leveling si identifica un algoritmo atto a massimizzare il ciclo di vita della flash, gestendone il consumo dei blocchi. Si hanno due varianti di wear leveling, uno dinamico e uno statico. Il *dinamic wear leveling* cerca di utilizzare in modo uniforme i vari blocchi che via via si rendono disponibili in modo da distribuire il carico su tutta la flash. Alcune memorie flash, oltre al numero massimo di cicli di cancellazione hanno anche un numero massimo di cicli di lettura tra due cancellazioni, quindi se i dati sono mantenuti troppo a lungo in un determinato blocco e questo è letto molte volte è possibile che i dati vadano persi. Lo *static wear leveling* tenta di risolvere questo problema spostando periodicamente i dati validi in un nuovo blocco. Il JFFS tratta il filesystem come una coda di log circolare e definisce una sola struttura dati per i file e le cartelle, il "raw inode". Il wear leveling che implementa è molto rigoroso e per alcuni versi non ottimale, a causa di un eccessivo movimento dei file sul dispositivo di memoria. Il file system JFFS2 nasce dall'esigenza di correggere questi problemi e di aggiungere nuove

11 Letteralmente livellamento del logorio

caratteristiche al JFFS, come la compressione e la possibilità di creare hard link. Il filesystem JFFS2, pur mantenendo la compatibilità con il suo predecessore è molto più flessibile. Permette l'uso di nuovi tipi di nodi, ognuno dei quali inizia con un header comune a tutti e contenente una bitmask, il tipo del nodo, la lunghezza totale del nodo e il CRC del nodo. Il file system è ora visto come una coda circolare di spazio disponibile, le scritture sono eseguite in modo sequenziale e ogni modifica ad un file comporta la sua completa riscrittura. I nodi del vecchio file sono marcati come “dirty nodes”, quelli del nuovo sono marcati invece come “clean nodes”. I nodi della coda che non sono mai stati utilizzati, o che sono stati puliti in seguito al garbage collection, sono marcati come “free nodes”. I nodi marcati come “clean”, “dirty” o “free” sono mantenuti in altrettante liste (*clean_list, dirty_list, free_list).* Le operazioni che effettua il JFFS2 sono simili a quelle compiute dal JFFS. I nodi sono scritti in modo sequenziale finché il blocco sulla flash non è riempito, a questo punto si prende un nuovo blocco dalla lista dei “free nodes” e si continua la scrittura del file (dei nodi) partendo dall'inizio del blocco. Quando lo spazio disponibile sta per finire è necessario recuperare dei blocchi richiamando il garbage collection. L'algoritmo implementato dal garbage collection è stato modificato. Per determinare da dove scegliere il blocco si usa un metodo probabilistico. Si calcola il modulo 100 del numero di *jiffies*, se il risultato è diverso da zero, il 99% dei casi, il blocco è scelto dalla lista dei nodi marcati come “dirty”, altrimenti il blocco è preso dalla lista dei nodi “clean”. Con questo metodo si riesce ad ottimizzare il garbage collection facendogli riutilizzare dei blocchi che erano già parzialmente usati e di tanto in tanto si sceglie un blocco dalla lista dei nodi “clean”. Quando il filesystem JFFS2 è montato vengono eseguite alcune operazioni. Si effettua una scansione del dispositivo, recuperando tutte le informazioni essenziali alla gestione del file system e calcolando il CRC di ogni nodo in modo da verificarne la validità. Quando la prima scansione è completa se ne effettua un'altra, in modo da costruire una mappa degli inode e permettere la cancellazione dei nodi marcati come obsoleti. Si cercano poi gli inode che non hanno link sul filesystem e si cancellano. Questo passo è ripetuto ogni volta che si cancella un inode relativo ad una

directory, alla ricerca di inode orfani. L'ultimo passo è relativo al rilascio della memoria allocata per costruire le strutture temporanee necessarie alla costruzione del file system.

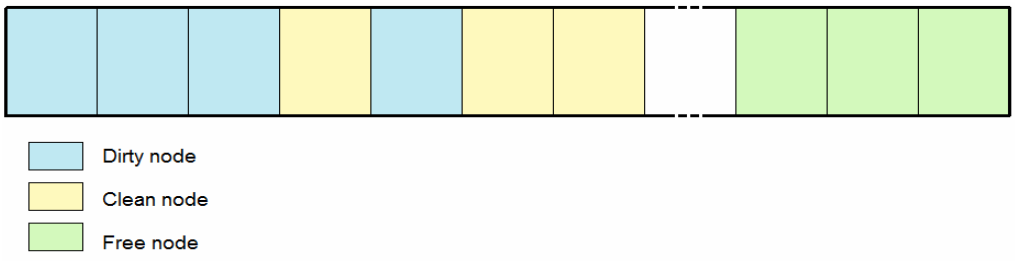


Figura 8: disposizione dei nodi nel filesystem JFFS2

All'inizio della vita del filesystem i nodi sono disposti come nella figura 8. I nodi marcati come “dirty” si alternano ai nodi “clean”, mentre i nodi “free” occupano la coda della lista. Quando si modifica un file, questo è completamente riscritto. Ciò è aderente alla definizione di wear leveling. Nei filesystem tradizionali, quando si modifica un file, si riscrivono solo i blocchi interessati dalle modifiche. Ma su una memoria flash ciò comporterebbe un diverso carico di operazioni di cancellazione/scrittura sui vari blocchi. Nel JFFS2 questo problema non si pone, dato che il wear leveling vuole che le scritture siano uniformi su tutto il dispositivo. Ciò può avvenire solo riscrivendo il file prelevando i

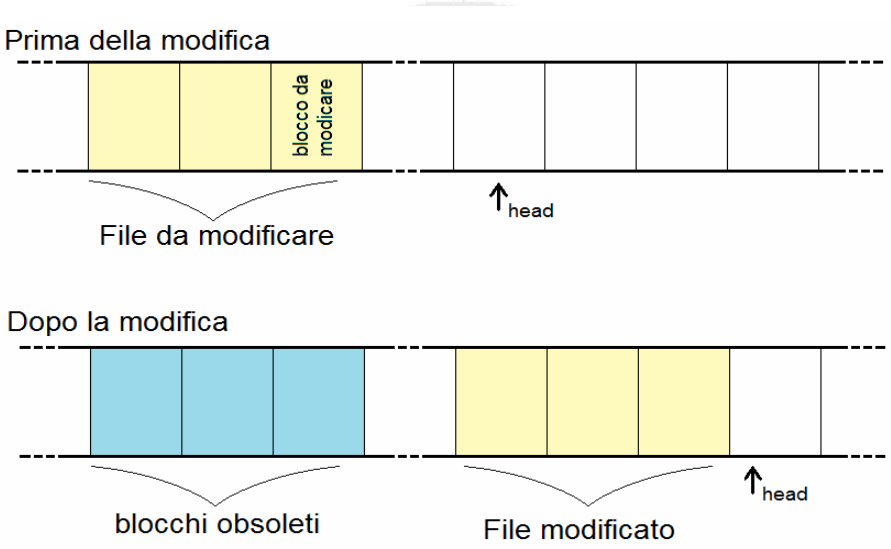


Figura 9: Nodi prima e dopo la modifica di un file

nodi necessari dalla testa della lista dei nodi free. I nodi che componevano il vecchio file sono marcati come obsoleti, e saranno recuperati e riutilizzati dalla prima esecuzione del

garbage collection (figura 9). Questo modo di operare è simile a quello dei filesystem con log. Una delle modifiche richieste per il futuro riguarda l'implementazione nel JFFS2 della feature *eXecute In Place* (XIP), in modo da poter eseguire il codice direttamente dalla flash. Attualmente, un programma per poter essere eseguito dalla CPU deve essere caricato dalla memoria in RAM. L'effettiva utilità della XIP è sotto esame, anche perché è in contrapposizione al fatto che i dati sulla JFFS2 sono compressi e quindi non possono essere eseguiti direttamente, ma necessitano di una decompressione in RAM prima di poter essere utilizzati. [6], [7], [20], [21], [22], [39]

1.6 Cross Compilation

I sistemi embedded, a causa delle limitate risorse di cui dispongono, non possono ospitare ambienti di sviluppo. Non è possibile eseguire su di essi nessun IDE, editor di testo, compilatori e debugger. Tuttavia è necessario scrivere applicativi per questi sistemi. Il problema è risolto utilizzando una macchina host opportunamente configurata per generare il codice oggetto che sarà poi trasferito ed eseguito sulla macchina target, il sistema embedded. Il processo relativo alla costruzione di un programma su di un sistema host per poter poi essere eseguito su di un sistema target è chiamato cross compilazione, l'elemento fondamentale della cross compilazione è il cross compilatore. Il gcc è stato portato su praticamente tutti i principali sistemi e per ognuno di essi è stato configurato per poter produrre dei binari ottimizzati per quella particolare architettura. Il gcc è il cross compilatore ottimale da utilizzare per costruire una cross toolchain. La costruzione di un cross compilatore e di una cross toolchain non sono operazioni semplici. Per effettuare una cross compilazione non è sufficiente disporre di un cross compilatore configurato ed ottimizzato per una particolare architettura hardware. Sono necessarie anche una serie di utility, che a loro volta devono essere costruite, ottimizzate e configurate per poter contribuire alla cross compilazione per quella particolare architettura. Il cross compilatore richiede il supporto delle librerie C e di altri eseguibili, come il linker, l'assembler, il debugger. L'insieme dei tool, delle librerie e dei programmi usati per la cross

compilazione si chiama *cross platform toolchain*, o *toolchain* in breve. Tutte le macchine atte alla compilazione di codice dispongono di una toolchain. Se gli eseguibili prodotti sulla macchina host dovranno girare su una macchina target dotata di una architettura simile all'host, allora la toolchain è detta *nativa*. Se macchina host e macchina target hanno differenti architetture allora la toolchain è detta *cross platform*. Il gcc utilizza un particolare prefisso per identificare la piattaforma per cui genererà i binari. Il prefisso ha la seguente forma CPUTYPE-MANUFACTURER-KERNEL-OPERATINGSYSTEM. Per un generico compilatore per PowerPc il prefisso standard può essere il seguente powerpc-unknown-linux-gnu. La cross toolchain può essere costruita a mano o si possono utilizzare dei tool (crosstool, buildroot, crossdev) che cercano di automatizzare tutto il processo di costruzione della toolchain. Il principale problema che si può incontrare utilizzando questi tool è che potrebbero non funzionare e fallire la costruzione della cross toolchain. Questa eventualità può accadere soprattutto se si cerca di utilizzare una versione del compilatore gcc, dei binutils, delle librerie C o del kernel che ancora non sono supportati dal tool. In questi casi l'unica alternativa è quella di costruire a mano la toolchain. La costruzione della toolchain può rivelarsi un compito più o meno complicato a seconda dei pacchetti software che si utilizzeranno per la sua realizzazione. I componenti fondamentali per costruire un compilatore e una toolchain, sia nativi che cross platform, sono i seguenti:

- Binutils: i cui sorgenti sono disponibili per il download da ftp://ftp.gnu.org/gnu/binutils. Comprende un set di tool, tra cui l'assembler as, il linker ld, programmi per la gestione dei file oggetto e delle librerie statiche e dinamiche (ar, ranlib, objcopy, nm)
- Compilatore Gcc: disponibile su ftp://ftp.gnu.org/gnu/gcc.
- Libreria C Glibc: è la libreria fondamentale a cui tutti i programmi C vanno linkati, i sorgenti sono disponibili su ftp://ftp.gnu.org/gnu/glibc. Oltre alla libreria C sono disponibili una serie di add-on che si potrebbero voler incorporare nella glibc.
- Sorgenti del kernel Linux: disponibili su http://www.kernel.org.
- Infine occorre verificare l'esistenza di patch per ognuno di questi pacchetti, che

vadano a risolvere dei problemi specifici per l'architettura target.

Oltre a definire per quale architettura si sta costruendo la toolchain, occorre procedere alla selezione della combinazione più appropriata del software che comporrà la toolchain, è necessario che la versione del kernel (gli header del kernel, necessari alla compilazione sia delle librerie C che del compilatore), del compilatore C, delle binutils e delle librerie C siano compatibili tra di loro. É necessario procurarsi le eventuali patch, necessarie alle proprie esigenze ed applicarle.

Sui sistemi Gnu/Linux uno degli strumenti più diffusi per la gestione delle varie fasi della costruzione di un software, dalla configurazione alla compilazione, sono gli Autotools (autogen, autoconf, automake, configure). Utilizzando gli autotools è possibile semplificare, per quanto possibile, la costruzione della cross toolchain. Lo script utilizzato per configurare l'ambiente di compilazione è *configure*. Di default questo script assume che la macchina host e quella target siano le stesse. Se le due macchine sono diverse, è necessario esplicitare il target passando allo script lo switch *--target,* o settando la variabile d'ambiente TARGET, seguito dal nome del sistema per il quale si sta generando il codice, ad esempio *--target mips-elf (export TARGET=mips-els)*. Quindi bisogna decidere dove installare la cross toolchain utilizzando lo switch --prefix o settando la variabile d'ambiente PREFIX, in modo che i cross binari prodotti non interferiscano con quelli della toolchain nativa, ad esempio --prefix=/tools/croos-compiler/mips-elf. É utile modificare la PATH aggiungendo il percorso in cui si installerà la cross toolchain. [6], [7], [25], [26], [34]

1.7 Dal boot al login

La sequenza di inizializzazione di un sistema Gnu/Linux è composta da una serie di fasi distinte che vanno dal momento in cui la macchina viene accesa al momento in cui all'utente viene chiesto di effettuare il login. La prima fase è comune a tutti i sistemi operativi ed è l'inizializzazione dell'hardware. Viene eseguito il codice contenuto nel

BIOS, il cui scopo è quello di verificare attraverso una serie di test (POST[12]), se la macchina è correttamente funzionante. Se i test effettuati sono superati positivamente si passa alla fase successiva. Il bios tenta di mandare in esecuzione il codice contenuto in alcune zone note di una serie di dispositivi. Storicamente il primo dispositivo da cui il bios tenta di effettuare il boot del sistema è il disco floppy, se presente, altrimenti tenta nell'ordine i dischi rigidi installati. Oggi il boot del sistema può avvenire anche da altri dispositivi, come i Cdrom/Dvd, da memorie flash installate su penne USB, dalla rete. Se il boot è eseguito da un disco (floppy o hard disk), quello che il bios esegue è il contenuto dell'MBR[13]. Il codice contenuto nell'MBR, detto stage1 boot loader è caricato dal primo settore del disco ed è grande soltanto 512 byte, quindi, a causa delle ridotte dimensioni, il solo compito che può svolgere è quello di caricare ed eseguire il vero boot loader, lo stage2 boot loader. L'obiettivo dello stage2 boot loader è quello di caricare in memoria ed eseguire il kernel. Il kernel, una volta caricato e mandato in esecuzione, si occuperà di effettuare altre inizializzazioni, di montare il root filesystem, di caricare i driver delle periferiche e quindi passerà il controllo al processo init, che terminerà la fase di start up della macchina portando alla richiesta di login. Il bootloader, quindi deve gestire una delle fasi più delicate dell'inizializzazione della macchina.

1.7.1 Il bootloader

In Linux si hanno a disposizione diversi boot loader, fra i quali i più utilizzati sono LILO (LInux LOader) e Grub (GRand Unified Bootloader). Entrambi si occupano dello stage 2, devono cioè caricare e mandare in esecuzione il kernel Linux. Lilo e Grub differiscono principalmente nel modo in cui sono gestite le modifiche al sistema. Lilo ha bisogno di essere reinstallato dopo ogni modifica al suo file di configurazione che descrive in quale partizione si trova e qual'è il kernel da caricare, mentre grub non ha bisogno di questo passaggio. Negli ultimi anni grub è diventato il bootloader di riferimento delle principali distribuzioni. Grub, come Lilo, può essere installato sia sull'MBR del disco di avvio sia sul

12 Power On Self-Test
13 Master Boot Record

boot record di una partizione. Grub nasce con il principale obiettivo di risolvere le deficenze di Lilo. Fornisce una shell, all'interno della quale è possibile eseguire una serie di comandi utili a modificare le impostazioni da passare al kernel durante il boot, o a caricare un kernel da rete o, ancora, a caricare un kernel non elencato nel menu di configurazione di grub, che è usualmente salvato in /boot/grub/grub.conf. Grub usa una speciale nomenclatura per identificare i dischi e all'interno di questi, le partizioni in cui cercare il kernel da eseguire. Le unità a cui si fa riferimento devono essere racchiuse da parentesi tonde, le periferiche sono identificate da una sigla (si usa hd per gli hard disk, indipendentemente dal fatto che siano IDE, SATA o SCSI, fd per i floppy disk) e da un numero cardinale progressivo. In base a queste regole il primo hard disk presente sulla macchina, che il kernel identificherà con il device /dev/sda, viene identificato come (hd0). Con lo stesso procedimento si ordinano le partizioni presenti su ogni disco, la prima partizione primaria è identificata dal numero zero, la prima partizione logica, indipendentemente dal numero di partizioni primarie presenti sul disco, è identificata dal numero 4. Quindi la seconda partizione del primo disco è per grub (hd0,1), la prima partizione logica del secondo disco è (hd1,4). Il comando utilizzato per installare grub è grub-install, l'argomento passato al comando identifica il disco su cui installare il bootloader. Per installare il bootloader sull'MBR di un floppy disk è sufficente dare uno dei seguenti comandi:

```
grub-install '(fd0)'
grub-install /dev/fd0
```

Il risultato ottenuto sarà che grub avrà scritto lo stage1 sul settore di avvio del dischetto, più una serie di altri stage in una apposita directory. Per installare lo stage1 sull'MBR del disco rigido, o su una partizione, è sufficente sostituire a fd0 la sigla che identifica la coppia disco, partizione:

```
grub-install '(hd0)'
grub-install '(hd0,10)'
```

il primo comando installa lo stage1 sull'MBR del primo disco, il secondo comando installa lo stage1 sul partition boot record della undicesima partizione del primo disco. Se si sta installando grub su un floppy o su di una partizione di un disco è necessario, prima di tutto, smontare il device. Si è visto che il bootloader è composto da più stage che vengono caricati in sequenza per portare poi all'esecuzione del kernel. I file che contengono gli stage sono, normalmente, salvati nella directory /boot/grub, eseguendo il comando ls dal suo interno si ottiene:

```
# cd /boot/grub/
# ls -l
totale 1276
-rw-r--r-- 1 root root   45 27 giu 12:23 device.map
-rw-r--r-- 1 root root 11768 12 lug 09:46 e2fs_stage1_5
-rw-r--r-- 1 root root 11528 12 lug 09:46 fat_stage1_5
-rw-r--r-- 1 root root 10776 12 lug 09:46 ffs_stage1_5
-rw------- 1 root root  1585 4 ago 17:47 grub.conf
...
-rw-r--r-- 1 root root 66003 11 apr 22:02 splash.xpm.gz
-rw-r--r-- 1 root root  512 12 lug 09:46 stage1
-rw-r--r-- 1 root root 110532 12 lug 09:46 stage2
-rw-r--r-- 1 root root 11040 12 lug 09:46 ufs2_stage1_5
-rw-r--r-- 1 root root 10376 12 lug 09:46 vstafs_stage1_5
-rw-r--r-- 1 root root 13016 12 lug 09:46 xfs_stage1_5
```

Oltre agli stage1 e 2 si osserva la presenza di una serie di file stage1_5. Il solo compito

che esegue lo stage1 è quello di caricare lo stage2 o uno degli stage1_5. Lo stage2 è il cuore di grub, si occupa di eseguire tutte le operazioni necessarie al caricamento del kernel e generalmente è posizionato su di un filesystem. A causa del poco spazio a disposizione dello stage1, tutto quello che si può fare è codificare al suo interno la locazione fisica dello stage2 o dello stage1_5, cioè lo stage1 non è in grado di identificare la struttura di un filesystem. Lo stage1_5 permette la gestione del boot da una periferica che differisce da quelle classiche (dischetto o hard disk) ed è un ponte tra lo stage1 e lo stage2, viene caricato da stage1 e poi caricherà lo stage2. Quindi, mentre stage1 riesce a leggere solo dei blocchi e richiede che al suo interno sia codificato l'indirizzo da cui leggere lo stage2, stage1_5 può identificare un filesystem (ad esempio reiserfs_stage1_5 riesce ad identificare un filesystem di tipo reiser) permettendo, a differenza di lilo, lo spostamento dello stage2 in una locazione diversa da quella in cui è stato installato senza per questo compromettere la corretta esecuzione del boot. Un'altra differenza tra lilo e grub è nella gestione del file di configurazione in cui sono descritte le partizioni di boot con i kernel disponibili. Lilo richiede necessariamente un file di configurazione, se invece grub non lo trova, farà partire una shell (figura 10) da cui impartire i comandi necessari al boot. I comandi impartiti sono gli stessi che si sarebbero scritti nel file di configurazione.

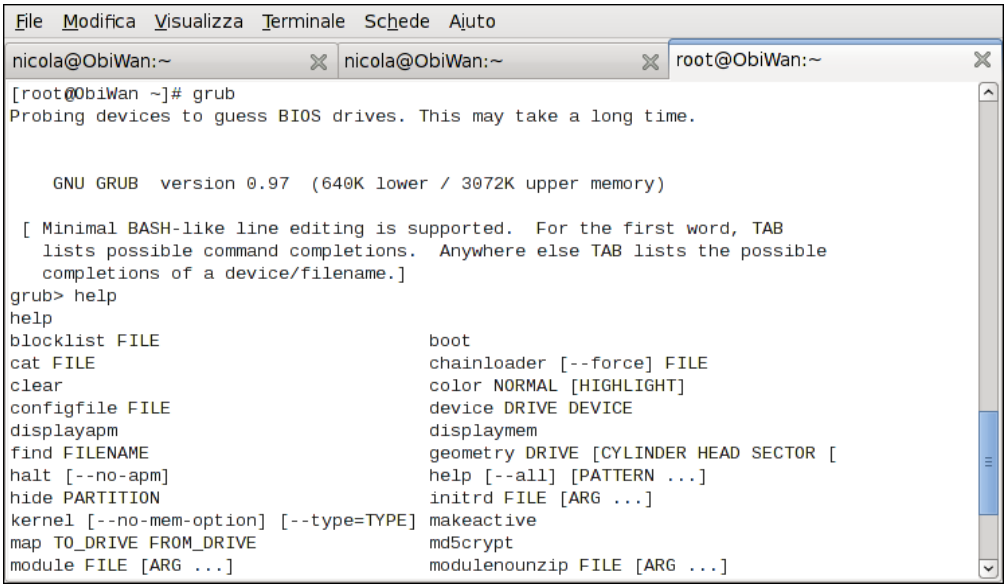


Figura 10: Shell di GRUB

Le informazioni assolutamente necessarie per permettere a grub di effettuare il boot, riguardano la locazione del kernel, del root filesystem, e se creata, dell'initrd. Un file di configurazione minimale di grub è simile a questo:

```
title kernel-2.x.y
root (hd0,2)
kernel /boot/vmlinuz ro root=/dev/sda3
initrd /boot/initrd-2.x.y.img
```

La prima riga definisce una stringa che verrà visualizzata in fase di boot e permetterà all'utente di selezionare da un menu quel particolare kernel. La seconda riga indica il disco e la partizione di root in cui cercare i file di stage (*1_5, 2). La terza riga imposta qual'è l'immagine del kernel da caricare definendone contemporaneamente il percorso, imposta inoltre, il percorso del root filesystem ed eventualmente una serie di parametri da passare al kernel. L'ultima riga imposta il percorso e il nome dell'init ram disk. [3], [31], [32]

1.7.2 L'inizializzazione del Kernel

Alla partenza (figura 11), il kernel effettua una serie di inizializzazioni che sono specifiche per la particolare architettura su cui sta girando, quindi inizializza i vari sottosistemi, monta il filesystem root e infine manda in esecuzione il processo init, che caricherà i servizi necessari e porterà alla richiesta di login. L'entry point del kernel è una routine assembly, generalmente codificata nel file arch/<name>/kernel/head.S, dove <name> indica la particolare architettura hardware su cui il kernel dovrà girare. Questa routine effettuerà varie operazioni, molte delle quali sono specifiche per il tipo di architettura. Tra le operazioni che svolgerà ci sono l'abilitazione dell'MMU, così che il kernel potrà utilizzare gli indirizzi virtuali e l'inizializzazione dello stack, permettendo l'invocazione della prima funzione in C, la start_kernel() implementata in init/main.c. La start_kernel() ha il compito di completare l'inizializzazione del sistema, invocando una lunga serie di

funzioni e terminando poi in un idle task. Dovrà richiamare la funzione setup_arch(), che effettuerà delle inizializzazioni specifiche per la piattaforma, come riconoscere il processore, la scheda su cui sta girando, esaminerà la lista dei parametri passati al kernel e inizializzerà la memoria. Quindi sarà chiamata la funzione trap_init() e poi la funzione init_IRQ(), che inizializzerà il sistema delle interruzioni. La funzione time_init() inizializza il clock del sistema, mentre la funzione console_init() fa partire un device in modo da redirigere tutti i messaggi che il kernel produrrà sullo schermo. Fra le ultime sarà chiamata la funzione calibrate_delay(). (in figura 11 e 12 il boot di EFML[14]).

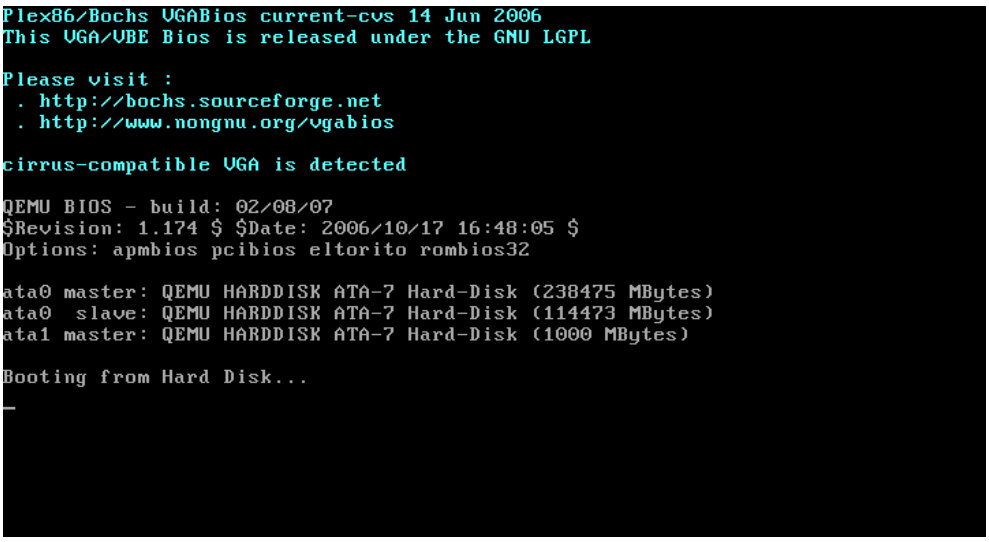

Plex86/Bochs VGABios current-cvs 14 Jun 2006
This VGA/VBE Bios is released under the GNU LGPL
Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/vgabios
cirrus-compatible VGA is detected
QEMU BIOS - build: 02/08/07
$Revision: 1.174 $ $Date: 2006/10/17 16:48:05 $
Options: apmbios pcibios eltorito rombios32
ata0 master: QEMU HARDDISK ATA-7 Hard-Disk (238475 MBytes)
ata0 slave: QEMU HARDDISK ATA-7 Hard-Disk (114473 MBytes)
ata1 master: QEMU HARDDISK ATA-7 Hard-Disk (1000 MBytes)
Booting from Hard Disk...

Figura 11: Boot del kernel di EFML in una macchina virtuale QEMU

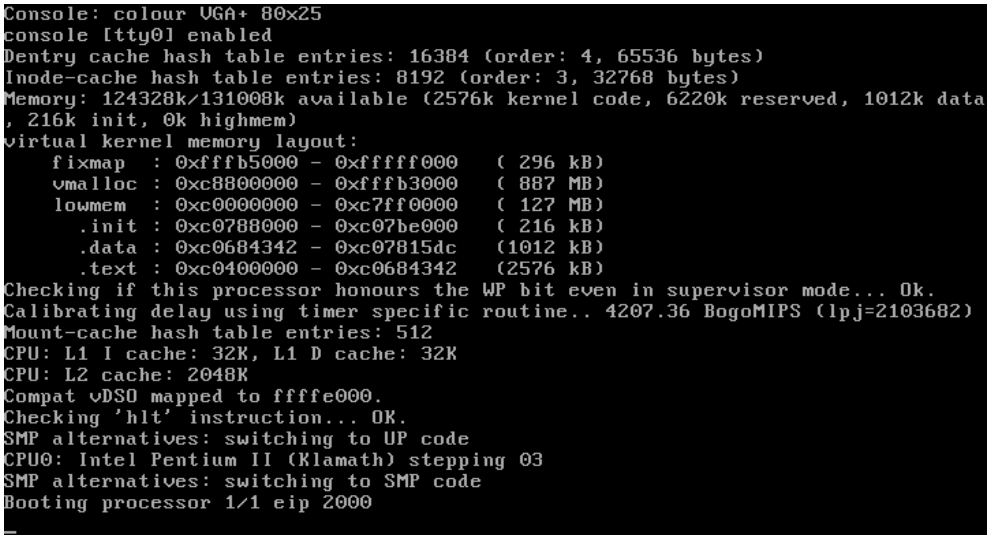

Console: colour VGA+ 80x25
console [tty0] enabled
Dentry cache hash table entries: 16384 (order: 4, 65536 bytes)
Inode-cache hash table entries: 8192 (order: 3, 32768 bytes)
Memory: 124328k/131008k available (2576k kernel code, 6220k reserved, 1012k data
, 216k init, 0k highmem)
virtual kernel memory layout:
fixmap : 0xfffb5000 - 0xfffff000 (296 kB)
vmalloc : 0xc8800000 - 0xfffb3000 (887 MB)
lowmem : 0xc0000000 - 0xc7ff0000 (127 MB)
.init : 0xc0788000 - 0xc07be000 (216 kB)
.data : 0xc0684342 - 0xc07815dc (1012 kB)
.text : 0xc0400000 - 0xc0684342 (2576 kB)
Checking if this processor honours the WP bit even in supervisor mode... Ok.
Calibrating delay using timer specific routine.. 4207.36 BogoMIPS (lpj=2103682)
Mount-cache hash table entries: 512
CPU: L1 I cache: 32K, L1 D cache: 32K
CPU: L2 cache: 2048K
Compat vDSO mapped to ffffe000.
Checking 'hlt' instruction... OK.
SMP alternatives: switching to UP code
CPU0: Intel Pentium II (Klamath) stepping 03
SMP alternatives: switching to SMP code
Booting processor 1/1 eip 2000

Figura 12: Boot del Kernel EFML in una macchina virtuale QEMU

14 Embedded FinMeccanica Linux

Terminata questa fase preliminare verranno inizializzati lo scheduler, il gestore della memoria e il Virtual File System (VFS). Al termine di queste operazioni il kernel, se necessario, carica l'immagine initrd, la decomprime, la monta in sola lettura e carica i driver necessari a montare il root filesystem, il filesystem principale del sistema. Tutti gli altri filesystem verranno montato ad esso. Il montaggio del root filesystem è uno dei momenti più importanti dello start up del kernel. Per poter montare il root filesystem è necessario aver caricato il driver che gestisce il particolare filesystem utilizzato per formattare la partizione su cui si trova il root filesystem. Tuttavia se non si monta il filesystem non è possibile caricare nessun driver. Per risolvere questo problema, i programmatori del kernel hanno creato il file initrd, che non è altro che un disco virtuale di memoria. Al suo interno initrd conterrà i driver necessari al montaggio del vero root filesystem e uno script, il linuxrc, che compirà tutte le operazioni necessarie affinché venga caricato il vero root filesystem. L'uso dell'initrd è superfluo nel caso in cui i driver necessari al montaggio del filesystem sono stati compilati all'interno del kernel. Una volta montato il root filesystem il kernel può eseguire il processo Init. I programmatori del kernel hanno sviluppato un nuovo metodo che sostituirà il vecchio initrd. Nei kernel della serie 2.6 è già possibile utilizzare initramfs per inizializzare il sistema. Initramfs è un archivio creato con l'utility cpio, che è estratto in un root fs al termine del caricamento del kernel. Una volta estratto l'archivio il kernel verifica la presenza dell'init e se lo trova gli affida il compito di completare l'inizializzazione del sistema attraverso il montaggio del root filesystem. Initramfs è linkato all'interno dell'immagine del kernel e dopo che ha svolto le varie operazioni di inizializzazione la memoria che occupa in ram può essere facilmente recuperata.

1.7.3 Init, il padre di tutti i processi

Init è il primo processo che il kernel manda in esecuzione, viene identificato nel sistema attraverso il PID 1 ed è, a sua volta, responsabile dell'avvio di tutti gli altri processi del sistema. Init dispone di un file di configurazione, /etc/inittab, all'interno del quale sono

indicate tutte le operazioni che dovrà compiere, tutti i processi, i demoni e i servizi da lanciare, per portare a termine la fase di boot ed effettuare l'inizializzazione del sistema. Init permette, quindi, la personalizzazione della fase di inizializzazione del sistema. Esistono due famiglie di sistemi che permettono la personalizzazione del sistema, SysV (System V) e BSD init. La differenza più appariscente tra i due metodi è relativa alla gestione dei runlevel. Nel BSD init esistono solo il *single user mode* e il *multi user mode*, e per completare il boot si passa per entrambi. Il SysV init, invece prevede diversi runlevel. Le principali distribuzioni Gnu/Linux utilizzano il SysV init, tranne la Slackware che preferisce utilizzare il BSD init. I runlevel sono delle particolari configurazioni in cui si può portare una macchina unix. Init leggerà il file inittab ed in funzione del runlevel in esso codificato, farà partire un insieme di processi, demoni e servizi invece che un'altro. I runlevel variano da 0 a 6 e come si evince dal contenuto del file inittab, sono i seguenti:

```
# cat /etc/inittab
# Default runlevel. The runlevels used by RHS are:
#  0 - halt (Do NOT set initdefault to this)
#  1 - Single user mode
# 2 - Multiuser, without NFS (The same as 3, if you do
  not have networking)
#  3 - Full multiuser mode
#  4 - unused
#  5 - X11
#  6 - reboot (Do NOT set initdefault to this)
#
id:5:initdefault:
```

I runlevel 0 e 6 sono utilizzati rispettivamente per spegnere e riavviare la macchina. Il runlevel 1 è utilizzato nel momento in cui si rendesse necessario effettuare la

manutenzione della macchina. Questa eventualità può verificarsi in presenza di grossi errori sul root filesystem, in questo caso il sistema partirà con una configurazione minimale e all'amministratore del sistema (il solo utente che avrà accesso alla macchina) verrà fornita una shell dalla quale potrà riparare il filesystem, utilizzando i classici strumenti come fsck, e quindi far ripartire il sistema. Il runlevel 2 abilita la multiutenza, permettendo a più utenti di effettuare il login alla macchina, ma non abilita la rete. Il runlevel 3 è simile al 2, consente a più utenti di loggarsi al sistema con la differenza che la rete è abilitata. Generalmente, se non si utilizza nessun ambiente grafico, questo è il runlevel di default. Il runlevel 4 non è usato e potrebbe essere utilizzato secondo le proprie necessità, creando una configurazione ad hoc. Il runlevel 5 è simile al runlevel 3, con la sostanziale differenza che il login avverrà in un ambiente grafico. Sarà quindi necessario caricare un server grafico X (X11 o Xorg).

Tutte le righe contenute nell'inittab o sono commenti ed iniziano con il simbolo di cancelletto, o hanno la seguente sintassi:

id: runlevel : azione : processo

- id, è una sequenza di 4 caratteri, per compatibilità con le versioni più vecchie di SystemV si usano solo due caratteri, che identifica la riga nel file
- runlevel, indica in quale runlevel sarà eseguita l'azione
- azione, indica l'azione da compiere, può assumere solo determinati valori da selezionare da un preciso insieme
- processo, infine, definisce quale programma lanciare e con quali parametri

Di seguito sono riportati alcuni tra i possibili valori che è possibile assegnare al campo azione:

- *boot*, il comando nel quarto campo è eseguito all'avvio, ignorando il runlevel
- *ctrlaltdel*, quando si preme la combinazione di tasti control alt del, init eseguirà il comando descritto nel quarto campo
- *initdefault*, usato per indicare ad init qual'è il runlevel di default
- *once*, il processo deve essere eseguito solo una volta, quando si entra nel runlevel

specificato

- *powerfail*, init deve eseguire il processo indicato nel caso in cui si verifichino dei problemi all'alimentazione della macchina
- *respawn*, se il processo termina deve essere riavviato
- *sysinit*, il processo indicato va eseguito al boot, indipendentemente dal runlevel
- *wait*, init deve attendere che il processo termini prima di continuare con il parsing dell'inittab

in base a queste regole, la seguente linea provoca il reboot immediato (-r now) della macchina, alla pressione dei tasti control alt e canc.

```
# Trap CTRL-ALT-DELETE
ca::ctrlaltdel:/sbin/shutdown -t3 -r now
```

Se si vogliono eseguire delle inizializzazioni generiche, indipendenti dal runlevel settato, allora si può usare una linea come la seguente (utilizzata dalle principali distribuzioni). Al boot (sysinit), init eseguirà lo script /etc/rc.d/rc.sysinit

```
# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit
```

rc.sysinit è uno script che si occupa di tutte le inizializzazioni di carattere generale, come l'impostazione della path e dell'hostname. Attiva lo swap, monta i filesystem virtuali come /proc e /sysfs, imposta i font. Controlla che i vari filesystem, durante il precedente shutdown, siano stati smontati correttamente altrimenti lancia fsck, attiva la rete, monta le partizioni leggendole dal file /etc/fstab, e tanto altro. Su alcuni sistemi, generalmente di derivazione Debian, il lavoro compiuto da rc.sysinit è, invece svolto da un altro script, lo rcS e la linea precedente è sostituita con una simile alla seguente:

```
si::sysinit:/etc/init.d/rcS
```

rcS eseguirà tutti gli script contenuti nella directory /etc/init.d aventi il nome che inizia con s o S. Terminata l'esecuzione di rc.sysinit (rcS), init prosegue con le inizializzazioni peculiari al runlevel settato. Tipicamente, nella directory etc/rc.d sono presenti due elementi fondamentali: lo script rc, e una serie di sottodirectory aventi nome rcX.d, dove la X indica un runlevel. Quindi si avranno le sottodirectory rc1.d, rc2.d e così via fino a rc6.d. Lo script rc si occuperà di avviare e fermare i vari servizi quando si cambia runlevel. Mentre le directory rcX.d conterranno dei link ad altrettanti script contenuti, in funzione della distribuzione, in /etc/rc.d/init.d o /etc/init.d. Tutti questi link avranno un nome che inizia per la lettera S o per K seguita immediatamente da un numero e quindi un nome esplicativo. La lettera S indica che quel link dovrà far partire (start) il servizio indicato, mentre la lettera K indica che il link interromperà (kill) il servizio. Il numero sta ad indicare in che ordine i servizi devono essere fatti partire o devono essere fermati. Se si volesse aggiungere uno script per attivare e interrompere un servizio, bisognerebbe seguire questi passi. Si crea lo script all'interno della directory /etc/rc.d/init.d, quindi nella sottodirectory indicante il runlevel si crea un link allo script. Il nome del link inizierà per Snn o per Knn, dove nn indica un numero cardinale, il primo attiverà il servizio, il secondo lo fermerà.

Per poter effettuare il login ed entrare nel sistema, init lancerà il programma getty, o una delle sue varianti disponibili sulle varie distribuzioni.

```
# Run gettys in standard runlevels
co:2345:respawn:/sbin/agetty xvc0 9600 vt100-nav
1:2345:respawn:/sbin/mingetty tty1
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
```

```
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

inittab lancerà getty su uno o più terminali virtuali, su più runlevel. Quindi la linea

```
2:2345:respawn:/sbin/mingetty tty2
```

indica che sarà lanciato il programma mingetty sul secondo terminale virtuale, per i runlevel 2, 3, 4 e 5. É possibile impostare il runlevel passandolo come parametro al kernel prima di effettuare il boot. Ad esempio utilizzando la possibilità offerta da grub di editare la riga usata per invocare il kernel si può fare qualcosa del tipo:

```
kernel /boot/vmlinuz ro root=path/to/root/filesystem 3
```

il numero 3 indica che si vuole effettuare il boot nel runlevel 3. Una volta terminato il lavoro da eseguire in quel runlevel, è possibile passare in un altro runlevel utilizzando il comando *telinit.* [32], [35], [36], [39]

1.8 Ottimizzazione dello spazio

Per i sistemi embedded il solo spazio disponibile per le memorizzazioni di massa è quello fornito dalle memorie flash. Nonostante il prezzo di questi dispositivi sia sempre più basso, è comunque utile cercare di razionalizzarne e ottimizzarne l'uso. Per ridurre l'occupazione di spazio si può intervenire su vari aspetti:

- si possono utilizzare filesystem compressi per l'archiviazione del kernel e delle applicazioni. Come si è visto nei paragrafi precedenti per i sistemi embedded basati su Gnu/Linux sono disponibili il Cramfs, SquashFs e JFFS2
- Si può configurare il kernel per rimuovere tutto il codice relativo ai driver, all'hardware e alle funzioni non necessarie.

- Si possono configurare le applicazioni utente e le librerie di sistema per eliminare le opzioni superflue.
- Si possono utilizzare dei software studiati e progettati appositamente per l'uso in sistemi embedded.

Il primo passo da compiere per ridurre lo spazio occupato dal kernel è quello di eliminare tutto il codice inutilizzato. Quindi si possono abilitare gli switch per le ottimizzazioni fornite dai vari compilatori. A compilazione effettuata si può processare tramite il comando strip l'eseguibile ottenuto, eliminando dal file oggetto tutto il codice e le strutture utilizzabili per il debug. Una gran quantità di spazio è occupata dalle librerie C. Le librerie C sono un componente essenziale di ogni sistema operativo, tuttavia spesso contengono codice ridondante o inutile per un sistema embedded, essendo state pensate, come la maggior parte degli applicativi GNU, per i sistemi desktop o per i server. Per recuperare un po' dello spazio occupato dalla librerie C, si può procedere in modo empirico rimuovendo le funzioni della libreria che non si utilizzeranno o che sono ridondanti, scelta comunque sconsigliata vista la complessità di questo tipo di software. Oppure si può scegliere di adottare delle librerie C appositamente studiate per i sistemi embedded. Esistono alcune librerie che sostituiscono la glibc sui sistemi embedded, tra le quali quella che, al momento, sembra essere la più completa è la libreria uClibc.

1.8.1 uCLibc

La libreria uClibc[15] è stata studiata originariamente per il progetto uClinux[16], una distribuzione linux pensata per i processori senza MMU (Memory Managment Unit). In seguito al successo ottenuto la libreria si è svincolata da uClinux, è stato aggiunto il supporto ai processori dotati di MMU ed è stata adottata da altri progetti. Attualmente uClibc supporta un gran numero di processori, può essere utilizzata come libreria condivisa, dato che per ogni architettura possiede un apposito loader. La uClibc deriva dalla glibc, quindi condivide molto del suo codice con la libreria C di Gnu. Molte delle

15 http://www.uclibc.org/
16 http://www.uclinux.org

funzioni e delle features raramente utilizzate sono state eliminate riducendo l'occupazione su disco e in RAM. La libreria uClibc è una scelta adatta per quei sistemi embedded che dispongono di risorse limitate ed hanno quindi bisogno che le dimensioni dell'eseguibile e dell'occupazione in memoria siano ridotte il più possibile. La glibc deve fornire l'infrastruttura per tutte le possibili applicazioni scritte in C, quindi porta con se una grossa quantità di simboli, funzioni, definizioni che per un sistema embedded non sono necessari. Come il kernel Linux, la libreria uClibc dispone di un comodo tool semigrafico da utilizzare per la sua configurazione (figura 15). Questa libreria insieme a BusyBox costituisce l'elemento portante di buildroot.

1.8.2 BusyBox

BusyBox[17] è stata scritta da Bruce Perens nel 1996 per la distribuzione Debian, attualmente è mantenuta da Erik Andersen, il manutentore di uClibc. Lo scopo del programma era quello di realizzare un sistema bootabile da un singolo dischetto da utilizzare sia come disco di ripristino che per l'installazione di Debian. Per poter essere usato come disco di ripristino, era necessario che il dischetto fosse capace di effettuare il boot e montare il filesystem presente sull'hard disk, inoltre doveva fornire gli strumenti necessari a riparare e ad amministrare il filesystem danneggiato. Attualmente BusyBox è uno dei componenti fondamentali di molti sistemi embedded basati su kernel Linux. BusyBox è un eseguibile che sostituisce molti dei comandi che si trovano installati di default in un sistema Gnu/Linux. Grazie alle sue ridotte dimensioni è la scelta ideale per i sistemi embedded, inoltre, utilizzando il suo meccanismo di configurazione, è semplice effettuare una scelta sui programmi che si desidera compilare e inserire nel sistema finale. I programmi disponibili in BusyBox, definiti applets, sono suddivisi per categorie, si hanno a disposizione applicativi per:

- Shell: ash, lash, ecc
- Utilità generali: cat, chmod, cp, dd, mv, ls, rm, ecc

17 http://busybox.net/

- Programmi per la gestione dei processi: ps, kill, ecc
- Utilità per la gestione dei moduli del kernel: insmod, rmmod, modprobe, lsmod, depmod
- Tool di sistema: reboot, init, syslogd, ecc
- Tool di rete: ifconfig, route, ping, telnet, wget, ecc
- Tool per gestire gli archivi: ar, cpio, gzip, tar, ecc

In realtà BusyBox fornisce un solo eseguibile, busybox, tutti gli altri programmi sono soltanto dei link simbolici a busybox. Il meccanismo che permette questa flessibilità risiede nel modo in cui gli argomenti scritti sulla linea di comando sono passati ad un eseguibile C. La funzione main di un programma scritto in C ha il seguente prototipo

int main (int argc, char argv[]);*

il vettore argv contiene l'elenco delle stringhe componenti il comando battuto al prompt. La prima di queste stringhe, argv[0], è il nome del programma che si vuole eseguire, seguito da eventuali argomenti e switch. Quando busybox viene eseguito, utilizzando il valore di argv[0], riesce a determinare quale funzione interna richiamare per espletare il compito richiesto. La scelta delle applets avviene richiamando menuconfig, che lancia un tool di configurazione simile a quello utilizzato dal kernel Linux. La compilazione si effettua richiamando il classico comando make. La minor occupazione di spazio si ottiene linkando BusyBox dinamicamente a uClibc.

Anche i sistemi embedded basati su Gnu/Linux, come tutti i sistemi unix, terminano la loro fase di boot con l'esecuzione del processo init basato sul SystemV. Tuttavia i sistemi embedded non hanno bisogno di tutta la flessibilità che può offrire SystemV, raramente sono sistemi multiutente, mentre generalmente necessitano soltanto di un programma che possa inizializzare il sistema. Busybox, tra le varie applets, fornisce un sostituto di init, appositamente studiato per le necessità dei sistemi embedded, e come tutte le applets, in realtà, il programma /sbin/init non è nient'altro che un link simbolico a busybox. Se busybox non trova nessun file inittab, procede ad eseguire una serie di operazioni di default consistenti nel definire le operazioni da effettuare nel caso del reboot del sistema,

dell'halt del sistema e nel restart di init. La principale differenza che si osserva tra il classico init di SystemV e l'init di busybox è il non uso, in quest'ultimo, dei runlevel. La sintassi delle righe in inittab segue le linee classiche

id : runlevel : action : process

Anche se la sintassi da utilizzare per il file inittab è identica a quella usata per SystemV, busybox semplicemente ignora i valori di runlevel. Il campo id in busybox ha un significato differente, indica il terminale virtuale in cui il comando sarà eseguito, se il processo non sarà eseguito in una shell interattiva il campo id può essere lasciato vuoto. Il campo process, indica il percorso del processo da eseguire, con gli eventuali parametri da passargli. Infine il campo action può assumere uno dei seguenti valori:

- *sysinit*, indica il percorso allo script di inizializzazione del sistema.
- *respawn*, indica che il processo deve ripartire nel caso in cui terminasse.
- *askfirst*, è simile a respawn, con la differenza che chiede all'utente prima di far ripartire il processo. Non è utile su sistemi embedded che non hanno la supervisione umana.
- *wait*, init deve aspettare che il processo abbia completato la sua esecuzione prima di continuare con il parsing dell'inittab
- *once*, il processo è eseguito una sola volta, init può continuare con il parsing senza attenderne il completamento
- *ctrlaltdel*, esegue il processo associato alla pressione dei tasti control alt del
- *shutdown*, esegue il processo indicato allo spegnimento del sistema
- *restart*, esegue il processo indicato se init è riavviato.

La procedura di inizializzazione di init compie, nell'ordine, i seguenti passi:

- setup del signal handler per init
- inizializzazione della console
- parsing dell'inittab
- esecuzione dello script per l'inizializzazione del sistema /etc/init.d/rcS (dal nome dello script di inizializzazione si risale alle origini Debian di BusyBox)

- esecuzione dei comandi che nell'inittab hanno azione di tipo wait (i comandi che bloccano il sistema finché il comando stesso non termina la propria esecuzione)
- esecuzione dei comandi che nell'inittab hanno azione once (i comandi che sono eseguiti una sola volta)
- esecuzione dei comandi che nell'inittab hanno azione respawn (i comandi che devono essere riavviati nel caso in cui terminassero prematuramente la propria esecuzione)
- esecuzione dei comandi con azione askfirst

[7], [24]

Capitolo 2

Embedded Finmeccanica Linux

Di seguito sono riportate le operazioni e le scelte compiute, insieme alle ragioni che hanno portato a tali scelte, per la realizzazione della distribuzione Embedded Finmeccanica Linux. In realtà le distribuzioni create sono state due. Una basata sulle librerie Gnu glibc, l'altra sulle librerie uClibc.

Dalle specifiche hardware si evince che la scheda di riferimento, Concurrent Technologies VP417, ha un'architettura x86 e monta un processore Intel dual core. Nelle specifiche software, inoltre, viene chiesto che la versione del compilatore gcc sia minimo pari a 4.2.0. Sulla macchina a disposizione, dotata di processore Intel dual core, è installata la distribuzione Fedora 9, la quale è dotata del compilatore gcc 4.3.0. Ad una prima analisi, quindi, il requisito software richiesto è soddisfatto, inoltre sembra non essere necessario costruire una toolchain per effettuare una cross compilazione, dato che sia la macchina host che la macchina target sono dotate della stessa architettura e dello stesso processore. Tuttavia, in seguito ad alcune ricerche si è potuto appurare che il compilatore 4.3.0 non è adatto alla compilazione del kernel. Da Linux&C numero 64: “Recentemente è stata rilasciata una nuova versione del Gcc, la 4.3.0. [...] Per l'architettura x86 è stata completamente riscritta la componente che si occupa della generazione di codice in caso di *block move* e *block set*, in parole povere, la gestione delle funzioni memcpy() e memset(). [...] Per conformarsi all'ABI, su x86 e x86_64 il gcc non inserisce più l'istruzione cld prima dell'esecuzione di operazioni sulle stringhe. In effetti l'ABI stabilisce che all'inizio di una funzione il *direction flag* non deve essere impostato, per cui non lo si può impostare in alcun modo, ad esempio tramite codice ASM, se poi non lo si resetta. Sono coinvolte le medesime funzioni, memset() e memcpy(), come pure memmove(), wmemmove(), bzero(), bcopy(), strcpy(), ecc. In pratica tutte le funzioni definite in /usr/include/string.h. Il fatto è che proprio quest'ultima novità comporta per il kernel linux e dei *BSD delle

implicazioni che val la pena di approfondire. [...] Se si volesse ricorrere ad una semplificazione, potremmo pensare alla memoria come se la si potesse rappresentare geometricamente con un segmento, i cui punti sono i blocchi di memoria per cui, preso un punto all'interno nel segmento, ci si può spostare in avanti o all'indietro. Il *direction flag (DF)* nelle operazioni sulla memoria è ciò che stabilisce se nel maneggiare questi blocchi ci si debba spostare in avanti o all'indietro. *Cld (clear direction flag)* è un opcode che si occupa appunto di resettare codesto flag. Il ragionamento è semplice, dal momento che si presuppone che il DF sia vuoto [...], allora non vale la pena prendersi la briga di cancellarlo con un automatismo. Peccato che gli sviluppatori del kernel, invece, rinfrancati dal vecchio comportamento del gcc non se ne siano mai preoccupati. Una funzione che viene eseguita senza resettare preventivamente il flag di direzione è, purtroppo, quella che nel kernel si occupa della gestione dei segnali. Se si compila il kernel con gcc 4.3.0, il signal handler di Linux viene invocato col flag DF impostato nello stato in cui si trovava al momento in cui è stato emesso il segnale che esso intercetta. La conseguenza immediata è il *leak* di un bit si stato dal processo user space in esecuzione al momento dell'emissione del segnale. Ovviamente tutte le versioni gcc precedenti risolvevano da sé il problema, poiché l'istruzione cld eseguita prima di qualsiasi operazione inline o sulle stringhe faceva si che si partisse da uno stato noto (cleared appunto). Per dirla in parole povere se un programma user space imposta il DF in modo che esegua una memcpy() o un memmove() all'indietro e in simultanea viene emesso un segnale, il signal handler eseguirà un memmove() all'indietro mentre invece crede di andare in avanti. Ecco quindi che dei blocchi di memoria verranno copiati o spostati nella direzione sbagliata e nel posto sbagliato. Si tratta di un esempio da manuale di corruzione. [...] La maggior parte dell'utenza, soprattutto chi utilizza Linux come desktop e non ha lunghi periodi di uptime, non avrà problemi usando il gcc 4.3.0. per compilare il proprio kernel, occorre però essere consapevoli che c'è già chi sta analizzando le possibili implicazioni in materia di sicurezza e possibili exploit del kernel sfruttando questo baco" [27]. Data l'impossibilità di utilizzare il compilatore gcc 4.3.0 per la compilazione del kernel è nata l'esigenza di creare una

toolchain basata su di una versione del compilatore C esente da tale problema.

2.1 Costruzione della toolchain

La costruzione di una toolchain è una operazione lunga, delicata e complessa, come si può evincere da “Linux from Scratch” [4], si è cercato quindi una soluzione che accorciasse il più possibile i tempi relativi al setup dell'ambiente da utilizzare per la compilazione del software richiesto. La distribuzione di riferimento di Finmeccanica, per la realizzazione della distribuzione Finmeccanica Linux, è Gentoo, mentre uno dei principali tool attualmente utilizzati per costruire un root filesystem per sistemi embedded è Buildroot. La prima scelta è caduta su buildroot, che offriva un ambiente più “user friendly”, con tutti gli strumenti necessari ad automatizzare la costruzione sia della toolchain che del root filesystem. Dall'uso di questo tool deriva la prima distribuzione realizzata. In seguito, grazie anche ad una contemporanea esperienza maturata sull'uso di Gentoo, si è potuto creare una seconda distribuzione. La distribuzione embedded linux richiesta è stata, dunque, realizzata utilizzando entrambi gli ambienti. L'elemento comune ad entrambe le distribuzioni è il kernel, con il relativo software applicativo, quello che le differenzia è la diversa libreria C utilizzata.

2.1.1 Buildroot

Buildroot è un tool utilizzato per lo sviluppo di sistemi embedded, è composto da un insieme di Makefile e di patch, utilizzate per generare sia la toolchain per la cross compilazione che il root filesystem. Anche se il sistema embedded utilizza un processore di classe x86, può essere utile l'utilizzo di buildroot principalmente per due ragioni. Buildroot costruisce una toolchain basata sulla libreria uClibc, riducendo sensibilmente l'occupazione di spazio se paragonato a quello richiesto dalla classica glibc ed automatizza la costruzione dell'intero root filesystem, scaricando, patchando, compilando ed installando tutto il software necessario. Buildroot dispone di un tool di configurazione simile a quello utilizzato dal kernel Linux (figura 13). Da linea di comando è sufficiente

lanciare il comando *make menuconfig*. Buildroot utilizza una serie di Makefile che definiscono le operazioni da compiere per costruire tutti pacchetti software seguendo il corretto ordine. La versione di buildroot utilizzata, uno snapshot del codice presente nel repository subversion, è la 0.10.0. Modificando alcuni file di configurazione di Buildroot e creando i Makefile nelle apposite directory, è possibile estendere le funzionalità di questo strumento. Dopo un'analisi del tool, si è provveduto alla modifica di alcuni file di configurazione, necessaria soprattutto per poter selezionare la corretta versione del kernel (versione aderente alle specifiche software).

Il primo file di configurazione modificato è toolchain/kernel-header/Config.in, il file è letto da menuconfig e le opzioni in esso impostate sono mostrate all'utente per la configurazione dell'ambiente.

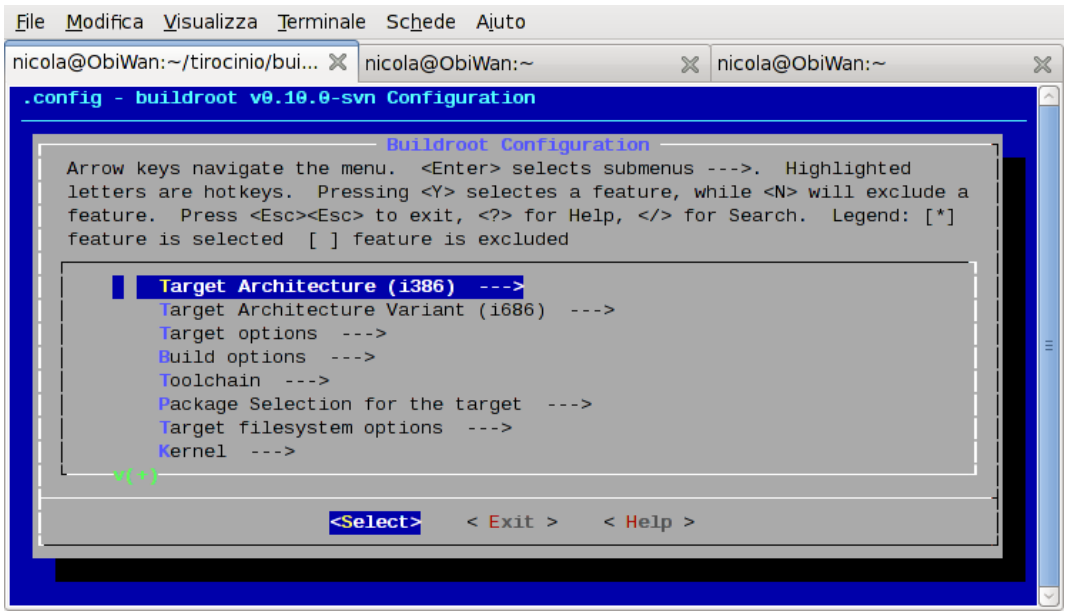

File Modifica Visualizza Terminale Schede Aiuto
nicola@ObiWan:~/tirocinio/bui...
nicola@ObiWan:~
nicola@ObiWan:~
.config - buildroot v0.10.0-svn Configuration
Buildroot Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> selectes a feature, while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is excluded
Target Architecture (i386) --->
Target Architecture Variant (i686) --->
Target options --->
Build options --->
Toolchain --->
Package Selection for the target --->
Target filesystem options --->
Kernel --->
<Select>
< Exit >
< Help >

Figura 13: Menu di configurazione di Buildroot

Il kernel richiesto nelle specifiche software è il 2.6.23 patchato con la patch rt13. Per evitare di dover patchare e configurare il kernel ad ogni successiva esecuzione del tool, è stato creato un tarball dall'albero dei sorgenti del kernel a cui sono state applicate le patch richieste e contenete, inoltre, il file .config riportante la configurazione finale. Tutte queste operazioni sono descritte in dettaglio nel paragrafo relativo al kernel. Quindi sono state apportate le seguenti modifiche al file toolchain/kernel-header/Config.in:

É stata modificata la voce di menu BR2_KERNEL_HEADERS_2_6_23. Nella versione di buildroot utilizzata, il kernel 2.6.23 è considerato deprecato, allora è stata commentata la linea in oggetto ed è stato settato il kernel come valido, inoltre si è modificata la stringa mostrata all'utente, esplicitando il fatto che il kernel è stato patchato.

```
BR2_KERNEL_HEADERS_2_6_23
#   depends on BR2_DEPRECATED
   depends on BR2_RECENT
#   bool "Linux 2.6.23.x kernel headers"
   bool "Linux 2.6.23 kernel headers (patch rt13)"
```

Il successivo file di configurazione modificato è stato target/linux/Config.in.advanced:
Da questo menu è possibile definire se e quale versione del kernel scaricare e compilare e se si vogliono applicare ulteriori patch. Le modifiche apportate sono relative all'elenco di opzioni presenti tra le voci BR2_DOWNLOAD_LINUX_VERSION e BR2_LINUX26_VERSION, alle quali si è aggiunto la linea *default "2.6.23" if BR2_LINUX_2_6_23*

```
config BR2_DOWNLOAD_LINUX26_VERSION
      string
      default "$(BR2_KERNEL_THIS_VERSION)" if BR2_KERNEL_BASE
      default "2.6.23" if BR2_LINUX_2_6_23
      default "2.6.21.5" if BR2_LINUX_2_6_21_5
      ...
config BR2_LINUX26_VERSION
      ...
      default "2.6.23" if BR2_LINUX_2_6_23
```

Dopo aver salvato i due file, sono stati compiuti una serie di cicli di configurazione/compilazione. Le principali configurazioni effettuate sono le seguenti:

dal sotto menu toolchain (figura 14) sono state abilitate le voci relative a large file support e wchar support, necessarie per la successiva compilazione di uClibc e Busybox (figura 15).

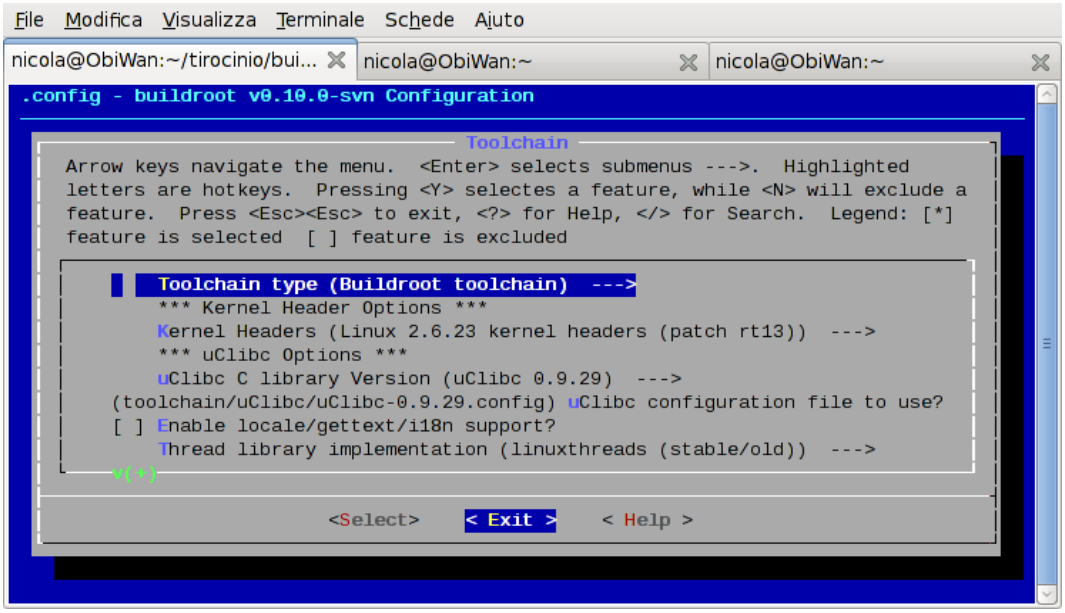

File Modifica Visualizza Terminale Schede Aiuto
nicola@ObiWan:~/tirocinio/bui...
nicola@ObiWan:~
nicola@ObiWan:~
.config - buildroot v0.10.0-svn Configuration
Toolchain
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> selectes a feature, while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is excluded
Toolchain type (Buildroot toolchain) --->
*** Kernel Header Options ***
Kernel Headers (Linux 2.6.23 kernel headers (patch rt13)) --->
*** uClibc Options ***
uClibc C library Version (uClibc 0.9.29) --->
(toolchain/uClibc/uClibc-0.9.29.config) uClibc configuration file to use?
[ ] Enable locale/gettext/i18n support?
Thread library implementation (linuxthreads (stable/old)) --->
<Select> < Exit > < Help >

Figura 14: Buildroot, il menu Toolchain

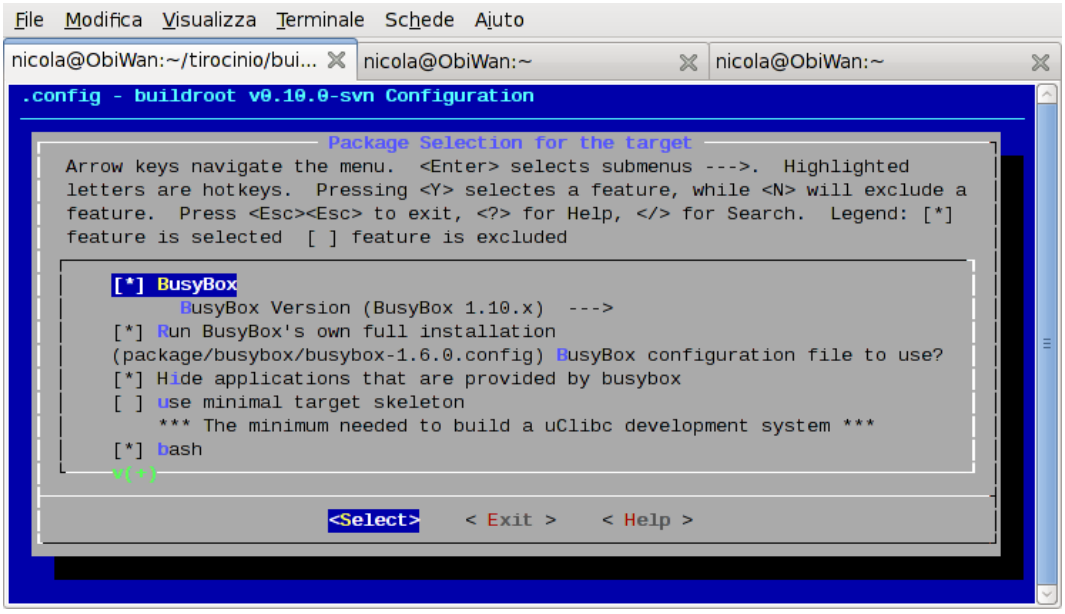

File Modifica Visualizza Terminale Schede Aiuto
nicola@ObiWan:~/tirocinio/bui...
nicola@ObiWan:~
nicola@ObiWan:~
.config - buildroot v0.10.0-svn Configuration
Package Selection for the target
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> selectes a feature, while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is excluded
[*] BusyBox
BusyBox Version (BusyBox 1.10.x) --->
[*] Run BusyBox's own full installation
(package/busybox/busybox-1.6.0.config) BusyBox configuration file to use?
[*] Hide applications that are provided by busybox
[ ] use minimal target skeleton
*** The minimum needed to build a uClibc development system ***
[*] bash
<Select> < Exit > < Help >

Figura 15: Sottomenu per la configurazione di BusyBox

Seguendo le specifiche software richieste si è selezionato la voce relativa alla compilazione del gdb server per il target, si è settato l'ambiente affinché compilasse busybox, bash, l'ntp e openssh. Le dipendenze necessarie a questi software sono risolte automaticamente, sfruttando le impostazioni definite nei Makefile. Dal sotto menu relativo al kernel è stata selezionata la voce indicante il kernel 2.6.23 con patch real time (la modifica apportata al file di configurazione), con gli altri settaggi si è imposto che buildroot costruisse l'immagine bzImage e che fosse installata nel root filesystem (figura 16).

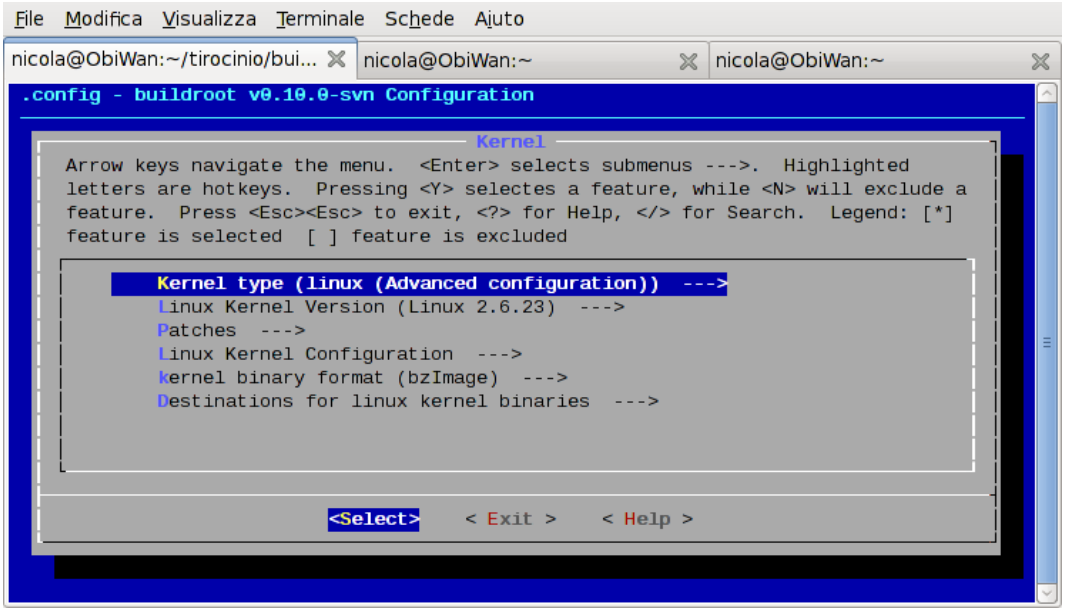

File Modifica Visualizza Terminale Schede Aiuto
nicola@ObiWan:~/tirocinio/bui...
nicola@ObiWan:~
nicola@ObiWan:~
.config - buildroot v0.10.0-svn Configuration
Kernel
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Y> selectes a feature, while <N> will exclude a feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature is selected [ ] feature is excluded
Kernel type (linux (Advanced configuration)) --->
Linux Kernel Version (Linux 2.6.23) --->
Patches --->
Linux Kernel Configuration --->
kernel binary format (bzImage) --->
Destinations for linux kernel binaries --->
<Select>
< Exit >
< Help >

Figura 16: Menu relativo alla gestione dei Kernel

Buildroot, oltre al menu semigrafico utilizzato per la propria configurazione, dispone di altri due menu simili, usati per la configurazione di busybox e di uClibc. Eseguendo da shell rispettivamente *make busybox-menuconfig* e *make uclibc-menuconfig*, si procede alla configurazione di busybox (figura 17) e uclibc (figura 18). I settaggi di rilievo effettuati per uClibc sono stati i seguenti:

- Dal sottomenu *general library settings*, abilitata la voce *enable large file support.*
- Dal sottomenu *network support*, abilitate le voci: *remote procedure call (RPC)*, (l'RPC è usato da NFS, Network File System) e *remote procedure call full rpc support*.

- Dal sottomenu *string and stdio support*, abilitata la voce *wide character support.*

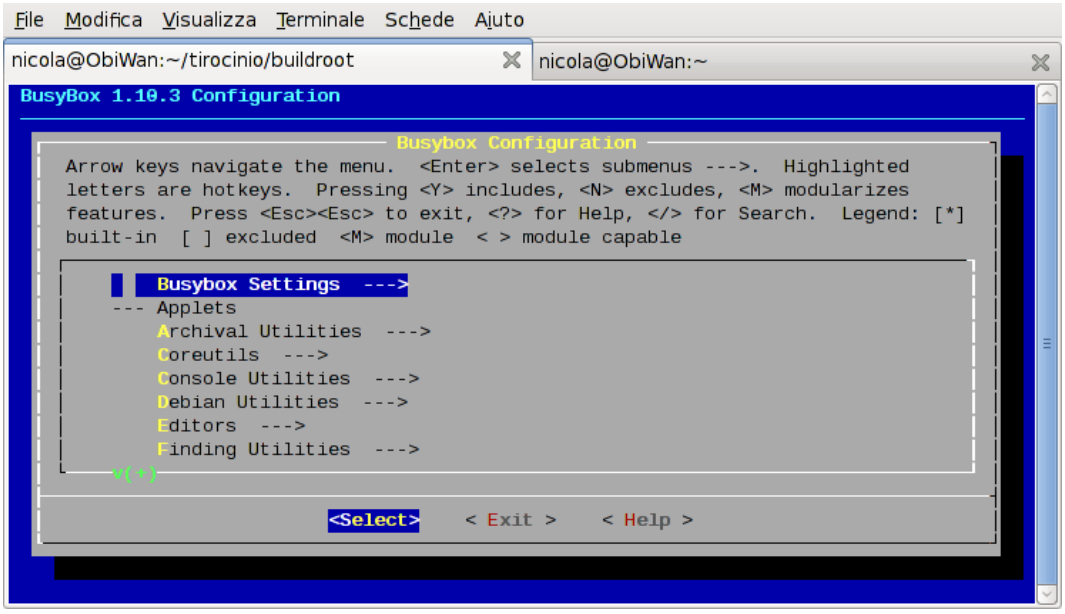


Figura 17: Configurazione di BusyBox

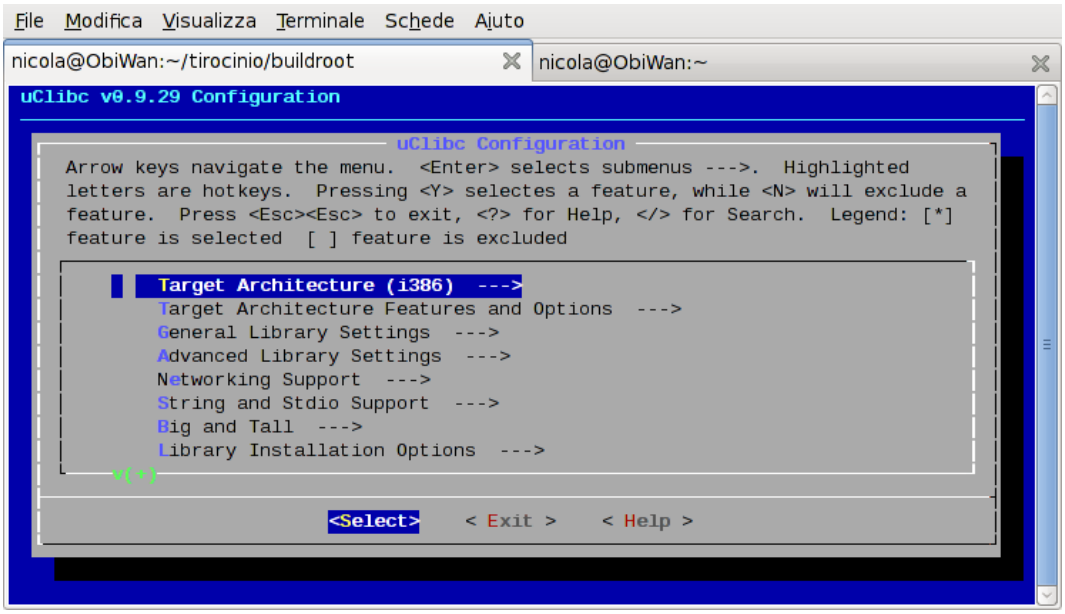

File Modifica Visualizza Terminale Schede Aiuto
nicola@ObiWan:~/tirocinio/buildroot
nicola@ObiWan:~
uClibc v0.9.29 Configuration
uClibc Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted
letters are hotkeys. Pressing <Y> selectes a feature, while <N> will exclude a
feature. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*]
feature is selected [ ] feature is excluded
Target Architecture (i386) --->
Target Architecture Features and Options --->
General Library Settings --->
Advanced Library Settings --->
Networking Support --->
String and Stdio Support --->
Big and Tall --->
Library Installation Options --->
<Select> < Exit > < Help >

Figura 18: Configurazione di uClibc

L'ultima configurazione è stata eseguita per modificare il Makefile relativo al pacchetto openssh (buildroot/package/openssh/openssh.mk). La versione di openssh impostata per la compilazione è stata settata a 5.0p1. Nel file openssh.mk si è modificato la path per

sysconfdir da /etc a /etc/ssh e si è editato il file S50sshd per riflettere le modifiche effettuate al sysconfdir, infine è stata modificata la patch openssh.patch, eliminando le modifiche da apportare al file sshd_config e applicandole a mano.

Terminate le varie configurazioni, eseguendo il comando make, buildroot ha provveduto a scaricare i pacchetti sorgenti e quasi tutte le patch necessarie, a creare la toolchain e il root filesystem. Il ciclo “configurazione pathing compilazione”, è stato ripetuto varie volte, per correggere gli errori che di volta in volta si sono manifestati, la maggior parte dei quali erano relativi alla mancata inclusione di qualche libreria o alla necessità di applicare ulteriori patch. Al termine dell'esecuzione buildroot, ha fornito in output i root filesystem che erano stati richiesti in fase di configurazione, tra i quali un root filesystem pronto per essere copiato su di una partizione della macchina di destinazione e una immagine iso utilizzata per verificare il boot del sistema su di una macchina virtuale. [37], [42]

2.1.2 Patch e configurazione del Kernel

Buildroot è stato configurato in modo tale da fargli utilizzare un tarball dei sorgenti del kernel preparato e configurato ad hoc. Lo stesso tarball è stato poi utilizzato per lo sviluppo della distribuzione basata sulle glibc. Di seguito sono riportate tutte le operazioni compiute per patchare e configurare l'albero dei sorgenti.

Al kernel vanilla 2.6.23, scaricato da kernel.org, sono state applicate sia le patch real time di MontaVista (patch-2.6.23.9-rt13.bz2), come richiesto dalle specifiche, sia le genpatches (genpatches-2.6.23-10.base.tar.bz2, genpatches-2.6.23-10.extras.tar.bz2), per poter disporre di un kernel gentoo compilant da utilizzare per un eventuale LiveCd. Dopo aver scompattato il kernel e i tarball contenenti le patch nella classica directory /usr/src e creato un link simbolico linux all'albero dei sorgenti, si è proceduto al patching del kernel utilizzando i seguenti comandi:

```
ln -s linux-2.6.23 linux
cd linux
```

```
## applicazione delle patch gentoo
for i in ../2.6.23/*; do patch -p1 < $i; done;
# applicazione delle patch real time di MontaVista
patch -p1 < ../patch-2.6.23.9-rt13
```

Dopo aver patchato il kernel sono stati aggiunti all'albero dei sorgenti i driver MTD forniti dal costruttore della scheda e contenuti nel pacchetto mtdmaps-1.10-01.tgz, scompattato in /usr/src. La procedura seguita è descritta su [9] pag. 7, ed è la seguente:

- Copia dei sorgenti del driver MTD nell'albero dei sorgenti del kernel

  ```
  cp -a /usr/src/mtd/map/cct/*.c linux/drivers/mtd/maps/
  cp -a /usr/src/mtd/map/cct/*.h linux/drivers/mtd/maps/
  ```
- Aggiunta la seguente linea alla fine del Makefile in linux/drivers/mtd/maps

  ```
  obj-$(CONFIG_MTD_CCTMAP) += mtd_map.o mtd_utils.o
  ```
- Aggiunte le seguenti istruzioni al file Kconfig in linux/drivers/mtd/maps/ prima della linea endmenu

```
config MTD_CCTMAP
      tristate “Concurrent Technologies MTD Map Driver”
      depends on X86 && MTD_PARTITIONS
      help
            Support for flash chips on Concurrent
       Technologies board
```

Nel file mtd_map.h sono descritte le partizioni che, si suppone, saranno create sulla memoria flash. Chi ha implementato il codice del driver ha assunto che la flash utilizzata fosse di 16Mb e che fosse stata divisa in 3 partizioni logiche. Tuttavia la flash che è montata sulla scheda di riferimento ha una capacità pari a 64Mb. Di conseguenza i valori settati nel sorgente devono essere modificati in funzione della diversa disponibilità di

spazio e in relazione alle partizioni che si intenderanno creare, e quindi ricompilare il kernel. Sul sistema finale, sarà necessario caricare il driver mtd (insmod mtdmap.ko) per poter selezionare e accedere la flash. Per i kernel della serie 2.6, la flash è selezionata automaticamente ed è smontata quando il driver è scaricato. Se il driver MTD è stato caricato e funziona correttamente, allora le partizioni potranno essere esaminate utilizzando il filesystem virtuale /proc.

L'ultimo aspetto riguarda la configurazione del kernel. Dopo aver dato i canonici comandi per pulire l'albero dei sorgenti (*make rmproper*), si è eseguito *make menuconfig*.

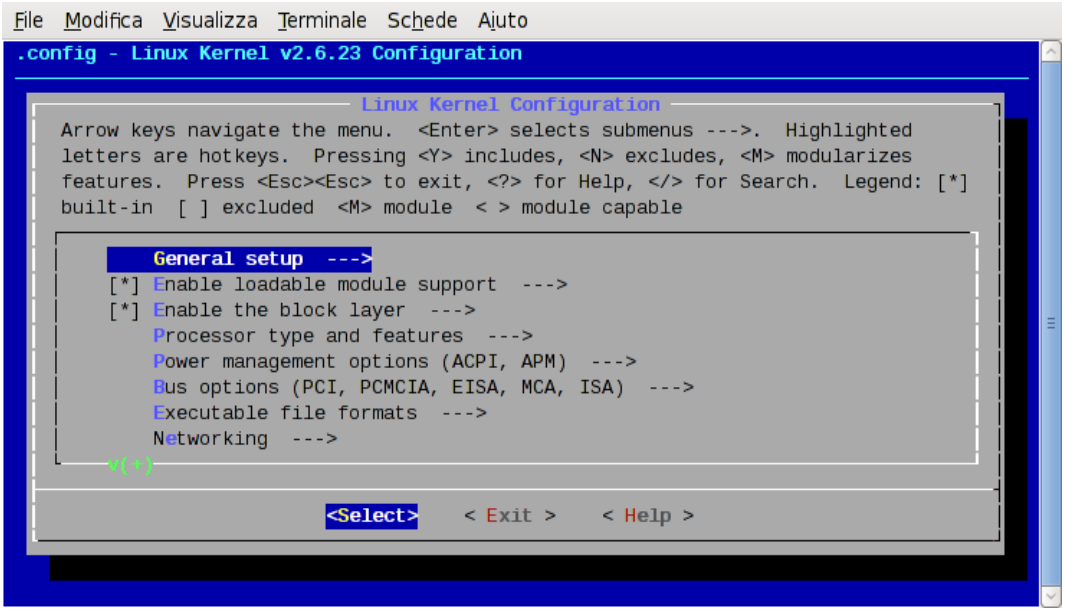

File Modifica Visualizza Terminale Schede Aiuto
.config - Linux Kernel v2.6.23 Configuration
Linux Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus --->. Highlighted
letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes
features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*]
built-in [ ] excluded <M> module < > module capable
General setup --->
[*] Enable loadable module support --->
[*] Enable the block layer --->
Processor type and features --->
Power management options (ACPI, APM) --->
Bus options (PCI, PCMCIA, EISA, MCA, ISA) --->
Executable file formats --->
Networking --->
<Select> < Exit > < Help >

Figura 19: Configurazione del Kernel

Le configurazioni effettuate, tenendo conto delle specifiche richieste e della particolare natura real time del kernel da realizzare, sono state le seguenti:

- dal menu General Setup
  - abilitate le Posix Message Queues: Le posix messages queue, fanno parte delle IPC. Nelle Posix message queues ad ogni messaggio è associata una priorità.
  - abilitata l'ottimizzazione dello spazio. In fase di compilazione, al compilatore verrà passato lo switch “-Os”, invece che “-O2”. Il risultato è un kernel più compatto.
  - abilitato “Configure standard kernel features (for small system)

- disabilitato “use full shmem filesystem”. Shmem è un filesystem interno, usato per gestire la memoria condivisa, può essere esportato in userspace sul filesystem tmpfs. Disabilitando questa opzione, shmem e tmpfs sono sostituite da ramfs che è la scelta consigliata per piccoli sistemi
- abilitato initial RAM filesystem and RAM disk (initramfs/initrd) support

- dal menu Processor type and features
  - abilitata la famiglia dei processori Core 2/Xeon
  - dal sotto menu “Preemption Mode (Complete preemtion (real time))”, selezionata “complete preemption”. Questa opzione riduce le latenze dello scheduling sostituendo gli spinlock utilizzati nel kernel con dei mutex prelazionabili. Con questa opzione abilitata, il kernel è immediatamente prelazionabile nel 95% dei casi, anche se sottoposto ad un carico inteso. Abilitando questa opzione si può costruire un sistema embedded o real time con tempi di latenza che sono garantiti essere minori o uguali a 100 usec.
  - Selezionata “Preemptible RCU”. Questa opzione riduce la latenza del kernel rendendo alcune sezioni RCU prelazionabili. La latenza è ridotta, però potrebbero verificarsi dei bug.
- dal menu “Power management options” è stato abilitato il supporto all'ACPI, come da specifiche software
- dal menu “Networking options” è stato abilitato il supporto al TCP/IP, come da specifiche software. Sono state disabilitate tutte le voci relative alle rimanenti tecnologie (IrDA, Bluetooth, Wireless, Radio)
- dal menu “Device Drivers”
  - abilitato il supporto per dispositivi SATA, SCSI, ATA/ATAPI, driver richiesti per poter gestire il disco esterno da utilizzare per il log.
  - abilitati i driver I2C, come da specifiche software
  - dal sottomenu “Memory technology Device (MTD) support”, come richiesto dal manuale di configurazione della scheda, sono state incluse nel kernel le

opzioni

- MTD concatenating support
- Direct char device access to MTD devices
- Caching block device access to MTD devices
- dal sotto menu “RAM/ROM/Flash chip drivers”, abilitate le opzioni
  - detect flash chips by Common Flash interface (CFI) probe
  - Il manuale della scheda indica di abilitare il supporto per i chip nand Intel/Sharp. Tuttavia durante i primi test effettuati nella sede dall'Mbda, avendo a disposizione la scheda si è potuto verificare che la memoria nand non veniva riconosciuta. Dopo svariati test si è appurato che il problema risiedeva nel fatto che il chip nand montato non è costruito da Intel/Sharp bensì da AMD/Fujitsu.
- dal sotto menu “Mapping drivers for chip access”, abilitato
  - support non-linear mappings of flash chips
- selezionato il sotto menu “NAND Device Support”

- dal sotto menu “Block devices”, sono stati selezionati
  - Loopback device support
  - Ram disk support
- selezionato il menu “Character Devices”
  - abilitato il supporto per “console on serial port” e “Unix98 PTY support”
  - abilitato il sotto menu “Watchdog Timer Support”
  - abilitato “Enhanced Real Time Clock Support”
  - abilitato “Real Time Clock Histogram Support”
- abilitato il supporto ai dispositivi USB
- abilitato il supporto al “Real Time Clock”. Il real time clock è settato per essere accessibile attraverso i filesystem /sys/class/rtc/rtcN (attraverso il sysfs), /proc/driver/rtc (attraverso il procfs) e /dev/rtcN (per i dispositivi a caratteri)
- da questo sottomenu sono stati disabilitati i driver multimediali (Audio/Video),

gli unici driver video abilitati sono relativi ai LED, al frame buffer, alle schede VGA e VESA.

- Abilitato il sotto menu “Hardware Monitoring support”. Le moderne schede montano una serie di sensori atti a monitorare la temperatura e/o il voltaggio di alcuni componenti (quali il processore, la scheda video, gli hard disk), o la velocità di rotazione delle ventole di raffreddamento.

- Dal menu “File systems”, il supporto ai filesystem ext2 e ext3 è stato incluso direttamente nel kernel. In tal modo è possibile evitare la creazione dell'initrd, anche se ciò ha come controparte la realizzazione di un kernel con una occupazione di memoria maggiore. La scelta di includere i driver direttamente nel kernel è stata dettata unicamente da fattori legati al tempo disponibile per la realizzazione della distribuzione.
  - dal sotto menu “Miscellaneous filesystems” è stato abilitato il supporto ai seguenti filesystem
    - JFFS2, con supporto alla compressione
    - Compressed ROM file system (CRAMFS)
    - SquashFS
    - NFS
    - tra gli ulteriori file system abilitati vi sono: CD/DVD (ISO 9660, UDF, Microsoft Joliet extensions), NTFS, pseudo filesystem (proc, proc/kcore, sysfs, shmfs)
- dal menu “Instrumentation Support”, sono state abilitate le opzioni per il profiling del kernel (Oprofile, come modulo e Kprobes compilato internamente)
- dal menu “Kernel hacking”: abilitato il supporto a:
  - wakeup latency timing, non-preemtible critical section latency timing, interrupts-off critical section latency timing
- abilitato il supporto alle API crittografiche, sono stati selezionati i seguenti algoritmi: SHA1, SHA256, SHA384, SHA512, ECB, Fcrypt, Blowfish, Twofish,

AES, Serpent, Deflate, CRC/CRC32c

I sorgenti del kernel, così patchati e configurati sono stati rimpacchettati in un nuovo tarball, usato per la creazione di entrambe le distribuzioni. Il pacchetto contenente il file .config ha permesso di saltare la fase di riconfigurazione durante le ripetute ricompilazioni, soprattutto in buildroot.

2.1.3 Gentoo

Anche per la realizzazione della distribuzione basata sulle librerie glibc, si è reso necessario creare una toolchain avente un compilatore gcc con numero di versione compreso tra 4.2.0 e 4.3.0. In questo caso il problema è stato risolto ricorrendo a Gentoo. Gentoo è una metadistribuzione GNU/Linux costruita utilizzando software libero. La sua caratteristica principale è quella di non utilizzare pacchetti binari precompilati, ma solo pacchetti sorgenti. Questa scelta garantisce un'alta flessibilità e permette di ottenere dei binari altamente ottimizzati per quel che riguarda le prestazioni. Il cuore di Gentoo è il suo sistema di gestione dei pacchetti, portage. É portage che permette di installare le applicazioni compilandole dal codice sorgente. Modificando le impostazioni definite nei file portage, l'utente può produrre eseguibili altamente ottimizzati per il proprio tipo di hardware. L'installazione di Gentoo va realizzata seguendo le istruzioni del ”Manuale Gentoo” [28]. Non si è cercato di effettuare l'installazione della distribuzione gentoo completa, bensì si è utilizzata la flessibilità offerta da gentoo per costruire nel minor tempo possibile una toolchain basata sul compilatore e sulle librerie C richieste. Dopo aver creato una apposita partizione, montata sotto la directory /media/gentoo, sono stati scaricati da uno dei mirror di gentoo i file *stage3-i686-2007.0.tar.bz2* e *portage-latest.tar.bz2*. Quindi si è seguita la procedura normalmente utilizzata per l'installazione della distribuzione Gentoo Linux. Il file stage3-*i686-2007.0.tar.bz2* è stato scompattato nella directory /media/gentoo con il comando:

```
mv stage3-i686-2007.0.tar.bz2 /media/gentoo
```

```
cd /media/gentoo
tar xvjpf stage3*.tar.bz2
```

ad operazione completata, è stato estratto all'interno della directory /usr il tarball contenente i file di portage, gli ebuild. Nei file ebuild sono descritti al programma portage tutti i passi da compiere per ottenere i binari dai sorgenti. Sono descritte le patch da applicare ai sorgenti, gli switch di configurazione, le directory in cui compilare e dove installare.

```
tar xvjf portage-latest.tar.bz2 /usr
```

Il sono file di configurazione modificato è stato /etc/make.conf al quale sono state aggiunte le seguenti linee

```
MAKEOPTS="-j2"
GENTOO_MIRRORS="ftp://ftp.unina.it/pub/linux/  \
     distributions/gentoo/"
ACCEPT_KEYWORDS="~x86"
```

MAKEOPT imposta un flag utilizzato da make, setta il numero di compilazioni che è possibile eseguire contemporaneamente in parallelo. GENTOO_MIRRORS imposta l'url ad un mirror della distribuzione gentoo. ACCEPT_KEYWORDS è settato in modo tale da permettere l'installazione anche di pacchetti che non sono ritenuti essere sufficientemente stabili.

Dalla distribuzione installata sulla macchina a disposizione è stato copiato il file etc/resolv.conf, contenente i nameserver per la rete. Sono stati, quindi, montati il filesystem /proc su /media/gentoo/proc e il filesystem /dev su /media/gentoo/dev con i seguenti comandi:

```
mount -t proc none /media/gentoo/proc
mount -o bind /dev /media/gentoo/dev
```

infine si è entrato nel nuovo ambiente effettuando il chroot e si è aggiornato il sistema:

```
chroot /media/gentoo /bin/bash
env-update
source /etc/profile
```

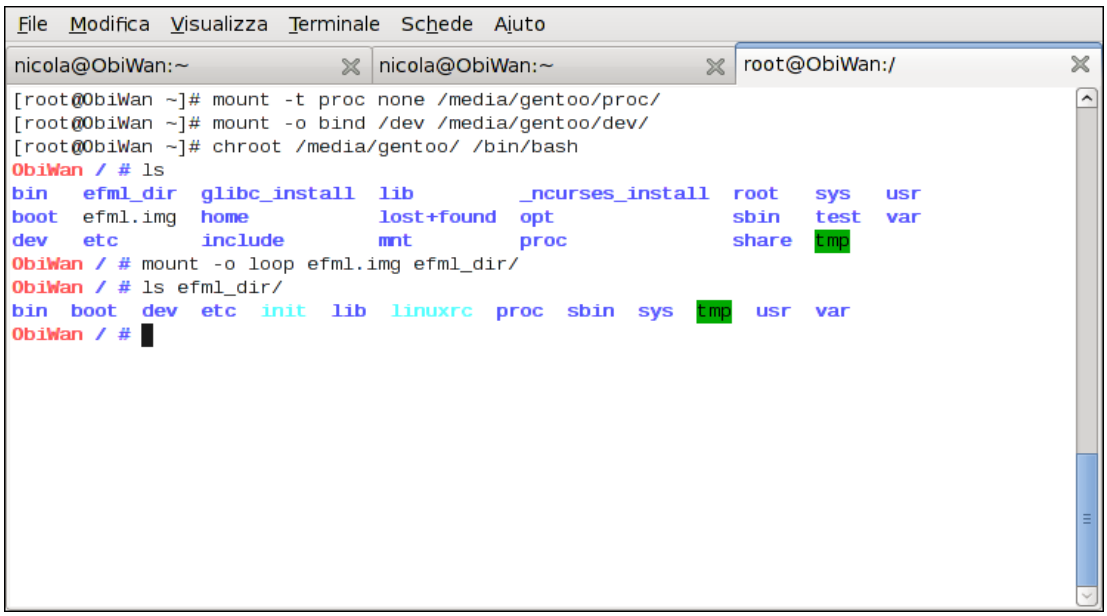

File Modifica Visualizza Terminale Schede Aiuto
nicola@ObiWan:~
nicola@ObiWan:~
root@ObiWan:/
[root@ObiWan ~]# mount -t proc none /media/gentoo/proc/
[root@ObiWan ~]# mount -o bind /dev /media/gentoo/dev/
[root@ObiWan ~]# chroot /media/gentoo/ /bin/bash
ObiWan / # ls
bin efml_dir glibc_install lib _ncurses_install root sys usr
boot efml.img home lost+found opt sbin test var
dev etc include mnt proc share tmp
ObiWan / # mount -o loop efml.img efml_dir/
ObiWan / # ls efml_dir/
bin boot dev etc init lib linuxrc proc sbin sys tmp usr var
ObiWan / #

Figura 20: chroot di Gentoo

da questo istante in poi, la procedura descritta sul manuale di installazione di Gentoo non è stata più necessaria. Le operazioni successive sono state finalizzate alla creazione della toolchain necessaria.

Bisogna dire che fra i vari progetti facenti parte della famiglia Gentoo, c'è anche Gentoo embedded [29]. Sul manuale relativo a gentoo embedded è descritta una procedura utilizzabile per l'installazione di una cross toolchain. L'operazione è basata, come tutto ciò che riguarda l'installazione di un software in gentoo, su portage e sul comando emerge.

Tramite emerge è possibili installare il programma *crossdev*, grazie al quale si può creare una toolchain rispondente alle proprie esigenze. Le operazioni compiute sono state:

```
emerge crossdev
```

che ha installato il tool, e quindi

```
crossdev --target i386-unknown-linux-gnu    \
         --b 2.18 --g 4.2.4                 \
         --k 2.6.23 --l 2.8
```

la sintassi di crossdev prevede che venga passato allo switch target una stringa che descriva la composizione del sistema di destinazione. In sostanza con questo comando si chiede la costruzione di un cross compilatore per architettura i386 su sistema linux che utilizzi le librerie C glibc (gnu). Le versioni dei vari software richieste sono:

- binutils 2.8 (--b 2.18)
- compilatore gcc 4.2.4 (--g 4.2.4)
- header del kernel 2.6.23 (--k 2.6.23)
- glibc 2.8 (--l 2.8)

Se uno degli switch relativi alla versione del software è omesso, crossdev utilizza di default l'ultimo pacchetto disponibile.

Purtroppo la procedura di installazione si interrompe prima che possa essere conclusa con successo. Un altro tentativo è stato fatto sostituendo alle librerie glibc, le uclibc. In questo caso il comando impartito è stato:

```
crossdev --target i386-unknown-linux-uclibc      \
         --b 2.8 --g 4.2.4 --k 2.6.23
```

ma come in precedenza, l'installazione è fallita.

L'ultima strada percorribile, prima di ricorrere alla creazione manuale della toolchain come descritto in Linux from Scratch, è stata l'installazione e compilazione dei singoli pacchetti tramite emerge.

Il primo passo è stato quello di installare gli header del kernel necessari alla successiva compilazione della toolchain. Gli header installati sono quelli del kernel 2.6.23.9 vanilla:

```
emerge =sys-kernel/vanilla-sources-2.6.23.9
```

Quindi è stato installato il compilatore gcc 4.2.4:

```
emerge =gcc-4.2.4
```

il compilatore è stato scaricato, patchato, compilato e installato con successo, a questo punto si sono installate le binutils e le glibc

```
emerge binutils-2.18
emerge glibc-2.18
```

anche in questi due casi la procedura di installazione dei software richiesti è andata a buon fine, completando in tal modo la realizzazione della toolchain necessaria allo sviluppo di Embedded Finmeccanica Linux basata sulla libreria glibc. In realtà durante la prima compilazione del Kernel utilizzando questa toolchain, si sono verificati alcuni errori. In particolare il comando xargs falliva durante la propria esecuzione. Da una ricerca in Internet si è appurato che era necessario aggiornare il pacchetto di cui il comando faceva parte. Il problema è stato risolto tramite l'emerge del pacchetto interessato:

```
emerge findutils
```

2.2 Preparazione delle partizioni

Come detto nel paragrafo 2.1.2, sarà necessario ricompilare il kernel dopo aver definito il numero e le dimensioni delle partizioni da creare sulla memoria flash ed aver modificato di conseguenza i file sorgenti del driver MTD. Effettuate queste operazioni e caricato il modulo nel kernel, eseguendo il comando cat /proc/mtd saranno visualizzate le partizioni presenti sulla flash, pronte per essere formattate. Il precedente comando restituirà l'elenco delle partizioni accompagnate da una serie di valori, simile al seguente:

```
dev:  size      erasesize  name
mtd0: 00180000  00020000   “MTD flash boot partition”
mtd1: 00740000  00020000   “MTD flash apps partition oone”
mtd2: 00740000  00020000   “MTD flash apps partition two”
```

I valori restituiti sono necessari per poter formattare correttamente il dispositivo. Come visto nel capitolo 1, il miglior filesystem utilizzabile per gestire una memoria flash è il JFFS2, tuttavia un device MTD non può essere formattato direttamente come JFFS2. Bisognerà seguire una procedura un po' più complessa per creare un filesystem JFFS2 su una delle partizioni della memoria flash. Sarà necessario scaricare, compilare e installare i sorgenti delle utility MTD[18], che permettono la creazione di un filesystem formattato come jffs2, quindi bisogna creare una directory di appoggio dalla quale si genererà una immagine formattata jffs2 che a sua volta verrà copiata sulla partizione scelta. Per poter compilare i sorgenti di *mtd-utils-1.1.0.tar.bz2*, è stato necessario installare la libreria lzo:

```
emerge lzo
```

una volta compilato i sorgenti di mtd-utils, gli eseguibili prodotti sono stati copiati sia

18 http://www.linux-mtd.infradead.org

nella directory /sbin del filesystem contenente la toolchain sia, in un secondo momento, nel filesystem della distribuzione embedded. Il sistema di sviluppo è stato quindi preparato per eseguire i seguenti passi, relativi alla procedura per creare l'immagine di un filesystem jffs2 partendo da una opportuna directory:

```
cd tmp
mkdir -p fs-mtdblock1/lost+found
mkfs.jffs2 -d fs-mtdblock1      \
           -e 0x20000           \
           -p 0x740000          \
           -o fs-mtdblock1.img
```

- -d ha per argomento il nome della directory da cui si genererà l'immagine
- -e vuole l'erase size della partizione, il valore è ricavato dall'esame dell'output di cat /proc/mtd
- -p vuole la dimensione della partizione, anche questo valore si ricava esaminando l'output di cat /proc/mtd
- -o è il nome dell'immagine

L'immagine fs-mtdblock1.img, non può essere copiata direttamente sulla partizione. Prima di tutto bisogna cancellare la partizione, e solo al termine di questa fase si può copiare l'immagine per poi montare la partizione

```
dd if=/dev/zero of=/dev/mtdblock1 bs=128k count=58
dd if=fs-mtdblock1.img of=/dev/mtdblock1
mount -t jffs2 /dev/mtdblock1 /mnt/mtdblock1
```

i valori da impostare per bs e count sono rispettivamente:

- bs è l'MTD device erase size
- count è ottenuto dividendo la dimensione della partizione per il device erase size (in questo esempio 0x740000 / 0x20000 = 3A in hex, 58 in decimale)

La partizione è inizializzata la prima volta che viene montata, è una operazione che può richiedere un certo tempo che è funzione della dimensione della partizione stessa e dal fatto che per poter ricreare il filesystem presente sulla flash sono necessarie diverse scansioni della stessa (cap. 1, par. 1.5). Una volta che la partizione è stata montata, è possibile utilizzarla come un normale dispositivo. Questa stessa procedura va, naturalmente, ripetuta per tutte le partizioni che si intendono creare sulla memoria a stato solido.

2.3 Immagine del root filesystem

Per creare il root filesystem della distribuzione si è scelto di non seguire la procedura suggerita dal manuale della scheda. Se si fosse seguita quella procedura si sarebbe dovuto creare una directory, all'interno della quale installare il software richiesto e poi, da questa, creare l'immagine jffs2 del filesystem root da copiare sulla flash. Si è invece scelto di creare preliminarmente l'immagine, formattata con il filesystem ext2 e montata in loopback su di una opportuna directory, in questa è stato installato il software richiesto. Questa scelta ha permesso di verificare immediatamente il rispetto della specifica relativa alla dimensione massima dello spazio occupabile dal root filesystem. In un secondo momento, quando si è potuto verificare che le varie fasi della costruzione della distribuzione erano andate a buon fine, si è proceduto alla creazione dell'immagine del filesystem da copiare sulla memoria flash, secondo la procedura descritta dal manuale che accompagna la scheda:

```
mount -t proc none /media/gentoo/proc
mount -o bind /dev /media/gentoo/dev
chroot /media/gentoo /bin/bash
```

```
dd if=/dev/zero of=efml.img bs=1024k count=50
mkfs.ext2 efml.img

mkdir efml_dir
mount -o loop efml.img efml_dir
```

tutto il software richiesto dalle specifiche e necessario alla creazione della distribuzione che si è compilato in seguito, è stato installato in /efml_dir.

2.3.1 Installazione del bootloader

Il manuale fornito a corredo della scheda, prevede la creazione di un floppy disk su cui installare il bootloader grub, e in seguito trasferire l'immagine del dischetto sulla partizione di boot della memoria flash. Per sopperire alla mancanza di un lettore floppy con cui creare il dischetto con installato il bootloader, si è creato un file immagine su cui installare grub. Come descritto precedentemente, è necessario creare le immagini delle directory formattate con il filesystem jffs2, da copiare poi sulla flash. Tuttavia, per poter effettuare i primi test e non disponendo della scheda su cui è montata la memoria flash, si è creata una immagine contenente un filesystem di tipo ext2. I comandi utilizzati sono i seguenti:

```
mkdir /mnt/tmp
dd if=/dev/zero of=fd.img bs=144k count=10
mkfs.ext2 fd.img

mount -o loop fd.img /mnt/tmp
grub-install --root-directory=/mnt/tmp '(fd0)'
```

con queste istruzioni si è creata la sottodirectory tmp in mnt ed è stato creato il file fd.img, che conterrà l'immagine del floppy, della dimensione di circa 1.44 Mb. Sul file immagine

è stata poi creato un filesystem ext2, che è stato montato in loopback sulla sottodirectory precedentemente creata. Per installare grub su questo filesystem si è usato il programma grub-install. Sull'immagine è stato creato l'albero di directory boot/grub. Spostandosi nella directory grub e dando il comando ls -l si ottiene il seguente output:

```
# ls -l
totale 246
-rw-r--r-- 1 root root   60 9 lug 19:31 device.map
-rw-r--r-- 1 root root 11768 9 lug 19:30 e2fs_stage1_5
-rw-r--r-- 1 root root 11528 9 lug 19:30 fat_stage1_5
-rw-r--r-- 1 root root 10776 9 lug 19:30 ffs_stage1_5
-rw-r--r-- 1 root root 10768 9 lug 19:30 iso9660_stage1_5
-rw-r--r-- 1 root root 12440 9 lug 19:30 jfs_stage1_5
-rw-r--r-- 1 root root 10984 9 lug 19:30 minix_stage1_5
-rw-r--r-- 1 root root 13376 9 lug 19:30 reiserfs_stage1_5
-rw-r--r-- 1 root root  512 9 lug 19:30 stage1
-rw-r--r-- 1 root root 110532 9 lug 19:30 stage2
-rw-r--r-- 1 root root 11040 9 lug 19:30 ufs2_stage1_5
-rw-r--r-- 1 root root 10376 9 lug 19:30 vstafs_stage1_5
-rw-r--r-- 1 root root 13016 9 lug 19:30 xfs_stage1_5
```

grub-install ha installato sull'immagine del disco lo stage1, lo stage2, essenziali per poter caricare il kernel, ed una serie di stage1_5. Ha inoltre creato un file (device.map) in cui sono state mappate le partizioni esistenti sui dischi della macchina di sviluppo.

2.4 Implementazione della distribuzione

Avendo a disposizione una toolchain funzionante e rispondente alle specifiche richieste (librerie C glibc compilant al compilatore gcc di versione superiore a 4.2.0) e dopo aver

costruito un file immagine in cui realizzare il root filesystem, si è proseguito con la compilazione dei software richiesti dalle specifiche, necessari all'implementazione della distribuzione. Ogni eseguibile generato dalla compilazione è stato passato come parametro al programma *ldd*, per poter determinare le dipendenze, sottaciute, richieste dai singoli software. Il dettaglio di queste operazioni è raccolto in appendice

2.4.1 Installazione del Kernel

Nel paragrafo 2.1.2 è stata descritta la creazione e configurazione del kernel. Il pacchetto ottenuto è stato scompattato in /usr/src, sono stati creati i canonici link simbolici e quindi si è proceduto alla compilazione del kernel. Sui sistemi x86 l'immagine del kernel da utilizzare è bzImage, su sistemi PowerPc o Alpha, l'immagine è vmlinux [30]. I moduli del kernel e il kernel stesso sono stati installati rispettivamente in efml_dir/lib e efml_dir/boot:

```
make mrproper
make

make INSTALL_MOD_PATH=/efml_dir modules_install
cp arch/i386/boot/bzImage /efml_dir/boot/vmlinuz-2.6.23
cp System.map /efml_dir/boot/System.map-2.6.23
cp .config /efml_dir/boot/config-2.6.23

ln -s vmlinuz-2.6.23 vmlinuz
ln -s System.map-2.6.23 System.map
ln -s config-2.6.23 config
```

2.4.2 NtpClient

Il pacchetto ntpclient utilizzato è stato ntpclient_2007_365.tar.gz. Come detto in precedenza, l'ntpclient fornisce un meccanismo atto a sincronizzare tra loro i client

presenti su reti di grandi dimensioni. Dopo aver scompattato i sorgenti e letto il file README allegato, si è proceduto alla compilazione del codice dando semplicemente il comando make. Al termine della compilazione l'eseguibile prodotto è stato copiato nella directory /efml_dir/bin:

```
cd ntpclient-2007
make
cp ntpclient /efml_dir/bin/
```

2.4.3 Bash

Bash è l'acronimo di Bourn Again Shell, è l'interprete di comandi testuale più utilizzato nei sistemi GNU/Linux. Il compito delle shell (interpreti dei comandi) è quello di permettere all'utente di comunicare ed impartire comandi al sistema operativo. Fornisce un semplice linguaggio di scripting, molto utilizzato per la scrittura di piccoli script. La versione di Bash installata è stata la 3.2. Il pacchetto bash-3.2.tar.gz è stato scompattato e ai sorgenti è stata applicata la patch bash-3.2-fixes-5.patch, necessaria per correggere alcuni bug scoperti dopo il rilascio del pacchetto [4].

```
patch -p1 < ../bash-3.2-fixes-5.patch
```

Il software è stato configurato come segue

```
./configure --prefix=/efml_dir/usr \
            --bindir=/efml_dir/bin \
            --without-bash-malloc \
            --enable-static-link
make
make install
```

Lo switch --without-bash-malloc fa in modo che bash utilizzi la funzione malloc() disponibile nelle librerie C glibc, invece della propria versione interna che può causare un segmentation fault [4].

Dopo aver installato il software sono stati cancellati il file bashbug* e tutti i file relativi alle localizzazioni (locale/ru). Dall'esame delle dipendenze è emerso che bash, per poter essere eseguito correttamente, richiede la libreria condivisa termcap, che fa parte del pacchetto ncurses. Si è reso necessario procedere alla compilazione e all'installazione dei sorgenti del pacchetto ncurses.

2.4.4 Gdb

Gdb o Gnu debugger è il debugger predefinito degli ambienti GNU. Permette di debuggare codice scritto in C, C++, ADA e Fortran per diverse piattaforme. Il pacchetto gdb installato ha il numero di versione pari a 6.6. Nelle specifiche software è richiesto che la distribuzione disponga dell'eseguibile gdbserver. Questo software non ha richiesto configurazioni particolari. É stato configurato, compilato e installato utilizzando i seguenti comandi:

```
.configure --prefix=/efml_dir/usr
make
make install
```

Nel processo di installazione sono stati copiati sulla directory di destinazione anche alcuni file, gdbtui, gdb, documentazione, non richiesti nelle specifiche software e che sono stati quindi cancellati per recuperare spazio sul disco.

2.4.5 BusyBox

Il programma busybox è stato ampiamente descritto nei paragrafi precedenti. La versione

di busybox utilizzata è stata la 1.10.3. Prima di compilare i sorgenti, sono state applicate le seguenti patch:

- busybox-1.10.3-tcpudp.patch
- busybox-1.10.3-udhcpc.patch
- busybox-1.2.2.1-max_host_len_40.patch

L'applicativo è stato configurato richiamando make menuconfig, le impostazioni di default sono state ritenute adeguate agli scopi prefissi. Le uniche personalizzazioni effettuate hanno riguardato l'abilitazione del comando alias e l'impostazione della path per l'installazione dei binari prodotti.

- Dal sotto menu “Installation Options” di “Busybox Settings”
  BusyBox installation prefix = /efml_dir

Dopo aver salvato le impostazioni fatte, si è proseguito con la compilazione e l'installazione impartendo i comandi:

```
make
make install
```

al termine della fase di compilazione si è ottenuto il seguente messaggio:

```
Trying libraries: crypt m
Library crypt is needed
Library m is needed
Final link with: crypt m
```

Sembrerebbe che busybox non sia stato linkato in modo corretto. Dopo aver effettuato alcune ricerche in rete, si è trovato che a questo problema è stato risposto nel seguente thread: http://www.mail-archive.com/busybox@busybox.net/msg02157.html

...

> but while compiling it is giving an error like

> Trying libraries: crypt m

> Library crypt is needed

> Library m is needed

> Final link with: crypt m

> What will be the reason for this.

It's not an error. Do you have busybox executable after this step?

If yes, then it worked.

...

L'eseguibile busybox è stato correttamente installato e analogamente sono stati creati tutti i link simbolici all'eseguibile, quindi come suggerito dalla risposta in mailing list, il messaggio d'errore è stato ignorato.

2.4.6 OpenSSh

Ssh è nato per permettere la comunicazione sicura tra due macchine. Ssh garantisce la sicurezza delle comunicazioni utilizzando una crittografia basata su chiave pubblica. OpenSSh[19] è l'implementazione libera del protocollo ssh, è sviluppata come parte del progetto OpenBSD[20] (una implementazione libera di BSD, la Berkeley Software Distribution, una variante di Unix sviluppata nell'università della California di Berkeley). La versione di OpenSSH installata è la 5.0p1. OpenSsh dipende dalle librerie zlib e openssl, per poter essere compilata è necessario averle installate entrambe. Le operazioni effettuate per la loro installazione sono descritte in seguito. Il pacchetto è stato configurato ed installato nel modo seguente. I file di configurazione di openssh saranno installati nella

19 http://www.openssh.org/
20 http://www.openbsd.org/

directory /etc/ssh.

```
./configure --prefix=/efml_dir/usr \
            --sysconfdir=/efml_dir/etc/ssh
make
make install
```

Al termine dell'installazione sono state generate le coppie di chiavi pubbliche e private rsa1, dsa, rsa

Generating public/private rsa1 key pair.
Your identification has been saved in /efml_dir/etc/ssh/ssh_host_key.
Your public key has been saved in /efml_dir/etc/ssh/ssh_host_key.pub.
The key fingerprint is:
3d:40:5a:ac:30:23:fd:f2:64:f6:13:dd:27:66:6c:b7 root@ObiWan
Generating public/private dsa key pair.
Your identification has been saved in /efml_dir/etc/ssh/ssh_host_dsa_key.
Your public key has been saved in /efml_dir/etc/ssh/ssh_host_dsa_key.pub.
The key fingerprint is:
45:f3:40:ec:7c:99:06:29:dc:6c:6a:2c:42:4f:79:21 root@ObiWan
Generating public/private rsa key pair.
Your identification has been saved in /efml_dir/etc/ssh/ssh_host_rsa_key.
Your public key has been saved in /efml_dir/etc/ssh/ssh_host_rsa_key.pub.
The key fingerprint is:
fc:dd:03:33:8e:00:1b:d9:da:c5:5b:63:b0:a3:66:7c root@ObiWan

2.4.7 Glibc

Durante il primo tentativo di installazione della libreria glibc è stato consumato tutto lo

spazio disponibile sull'immagine, è stato quindi necessario ripetere la compilazione eliminando alcuni componenti ed effettuando una installazione selettiva delle librerie condivise. Le librerie C utilizzate sono state le glib-2.8_p20080602. Sono state applicate le stesse patch utilizzate da gentoo. Secondo le modalità indicate nel file INSTALL, si è creata una apposita directory in cui effettuare il build della libreria. Inoltre è stato necessario abilitare la compilazione per i soli processori i686, in quanto i 386 non sono più supportati. Il pacchetto è stato configurato per installare le librerie prodotte sotto la directory glibc_install, da cui in seguito sono stati copiati i file necessari alla distribuzione nelle corrispondenti sotto directory di efml_dir. In questo modo è stato possibile evitare di installare gli header, la documentazione e tutti gli altri file necessari ad un sistema di sviluppo o desktop, ma non utili su di un sistema embedded. Si sono eseguite queste istruzioni per configurare e compilare i sorgenti.

```
mkdir -v ../glibc-build
cd ../glibc-build
echo "CFLAGS += -march=i686 -pipe -O2 -fno-strict-aliasing" \
     > configparms
../glibc-2.8-20080602/configure             \
     --prefix=/glibc_install                \
     --disable-profile --enable-add-ons     \
     --enable-kernel=2.6.0 --without-gd     \
     --without-selinux

make
make check
make install
```

gli switch utilizzati hanno le seguenti funzioni:

- disable-profile: non compila nelle librerie le informazioni per il profilig
- enable-add-ons: abilita la compilazione di add-ons, di pacchetti aggiuntivi. Se lo switch non ha nessuna lista di argomenti, allora saranno compilati tutti gli add-on che saranno trovati nella directory dei sorgenti.
- Enable-kernel: specifica la più piccola versione del kernel linux che la libreria deve supportare.
- Without-gd: disabilita la compilazione di memusagestat
- without-selinux: disabilita il supporto a selinux

2.5 Risoluzione delle dipendenze ed altro software di utilità

Per poter compilare alcuni dei software precedenti è stato necessario risolvere alcune dipendenze. In particolare:

- bash richiede che sul sistema siano presenti le librerie termcap, che fanno parte del pacchetto ncurses
- openssh, richiede la presenza delle zlib e delle librerie openssl.

Per poter garantire la corretta esecuzione del software richiesto dalle specifiche si è proceduto all'installazione dei pacchetti che risolvevano le dipendenze. Oltre a questi sono stati installati altri pacchetti la cui presenza può essere utile sulla distribuzione:

- kbd, per poter utilizzare una tastiera diversa da quella americana, l'unica disponibile con busybox.
- mtd-utils, contenente una serie di tool utili per effettuare operazioni di amministrazione (copia, cancellazione, test) sulla memoria flash.
- lzo, le librerie lzo sono necessarie al corretto funzionamento delle mtd-utils, in particolare all'eseguibile mkfs.jffs2.

2.5.1 Zlib

Il pacchetto zlib utilizzato ha numero di versione 1.2.3. Zlib implementa una libreria di compressione general purpose. Il formato di file implementato in queste librerie è descritto

negli RFC 1950, 1951 e 1952. L'rfc 1950 descrive il formato di zlib, l'rfc descrive come decomprimere il formato, infine l'rfc è relativo al formato gzip. Seguendo le istruzioni contenute nel file README e in testa al Makefile si è proceduto come segue:

```
make clean
./configure -s –prefix=/efml_dir/usr
make
make install
```

lo switch -s è utilizzato per richiedere la produzione della libreria condivisa libz.so.1.2.3

2.5.2 OpenSSL

Le OpenSSL sono una implementazione libera dei protocolli SSL v2/v3 (Secure Sockets Layer) e TLS v1 (Transport Layer Security) e di una serie di librerie crittografiche general purpose. Tra i vari algoritmi crittografici implementati figurano: DES, Blowfish, IDEA. Permette la generazione di message digest impiegando gli algoritmi di one-way hashing MD5, SHA, MDC2. Le coppie di chiavi pubbliche e private possono essere generate utilizzando gli algoritmi RSA, DSA, Diffie-Hellman. Il pacchetto utilizzato ha numero di versione 0.9.7m. Le istruzioni di compilazione contenute nel file INSTALL, invitano ad eseguire il comando make test per verificare la corretta compilazione del pacchetto. Le istruzioni eseguite sono le seguenti:

```
./config --prefix=/efml_dir/usr shared
make
make test
make depend
make install
```

2.5.3 Ncurses

I sorgenti delle librerie ncurses compilati hanno numero di versione 5.6. Ncurses (New curses) è una libreria per la gestione del display di una applicazione su terminale a caratteri. Include una serie di API per la gestione del mouse e per il supporto ad applicazioni semigrafiche. Dalla lettura del file INSTALL, che accompagna il pacchetto si è deciso si configurare il sorgente utilizzando i seguenti switch

```
./configure --prefix=/_ncurses_install \
      --disable-overwrite --with-shared \
      --without-normal --without-debug  \
      --with-libtool --disable-database \
      --disable-home-terminfo      \
      --with-fallbacks=linux,vt100,xterm
make
make install
```

funzione degli switch:

- with-shared: abilita la creazione di librerie condivise
- disable-overwrite: evita conflitti con le librerie già presenti sul sistema di sviluppo
- without-normal: non genera di default le “normal libraries” (le librerie statiche)
- without-debug: non genera le “debug-libraries”
- with-libtool: utilizza le libtool per creare le librerie condivise
- disable-database: le ncurses generalmente leggono i dati terminfo e termcap dal disco. Questo switch forza la libreria ad utilizzare una base di dati buil-in. Utili per i sistemi embedded che non necessitano di database esterni
- disable-home-terminfo: disabilita l'uso di $HOME/.terminfo
- with-fallbacks: specifica una lista di descrizioni di terminali da compilare nelle librerie.

La directory di installazione è stata impostata su _ncurses_install, questo perché non tutti i file installati sono di interesse per il sistema finale. In questo modo si è potuto individuare con più facilità quali tra i file appartenenti al pacchetto copiare sulla directory finale. Sono state copiate le librerie installate in _ncurese_install/lib in efml_dir/lib.

2.5.4 Kbd

Il pacchetto kbd contiene i file con le mappe delle tastiere e di alcuni font, contiene inoltre le utility necessarie alla gestione di font e keymaps. I sorgenti utilizzati hanno numero di versione 1.12. La configurazione e la compilazione sono state effettuate, come quasi tutti i software gnu, con la sequenza di comandi configure, make, make install:

```
./configure --prefix=/efml_dir
make
make install
```

Le mappe delle tastiere e i file dei font sono stati installati sotto la directory /lib/kbd.

2.5.5 Mtd-utils

Il pacchetto mtd-utils contiene una serie di tool da utilizzare per la gestione e l'amministrazione delle memorie flash. I sorgenti usati hanno numero di versione 1.1.0. Questo pacchetto non contiene nessuno script di configurazione ma è presente il solo Makefile. La compilazione si effettua richiamando il comando make. Al termine della compilazione, i binari creati sono stati copiati sia nella directory /sbin del filesystem contenente la toolchain, sia nella directory /sbin del filesystem della distribuzione. I binari installati sulla toolchain saranno utilizzati per creare creare il definitivo root filesystem della distribuzione, formattato con jffs2, mentre quelli installati sull'immagine della distribuzione potrebbero essere utili per la gestione futura del filesystem.

2.5.6 Lzo

LZO, acronimo di Lempel-Ziv-Oberhumer, è una libreria per la compressione e decompressione di dati in real time, l'algoritmo implementato è lossless, senza perdita, cioè i dati non subiscono modifiche o perdite di informazioni durante il processo di compressione. La libreria lzo è stata compilata per la distribuzione EFML in seguito all'installazione delle mtd utils, in quanto il programma mkfs.jffs2 ha tra le dipendenze la libreria lzo. I sorgenti utilizzati hanno numero di versione 2.03, la libreria è stata compilata ed installata con i seguenti comandi:

```
./configure --prefix=/efml_dir --enable-shared
make
make install
```

è stato necessario utilizzare il flag --enable-shared per poter generare la libreria condivisa richiamata da mkfs.jffs2.

2.6 Configurazioni e installazione

Terminata la fase di installazione dei software richiesti dalle specifiche e di quelli necessari a risolvere le varie dipendenze, si è passati alla configurazione della distribuzione. L'attenzione è stata rivolta agli script di inizializzazione e ai file di configurazione contenuti nella directory /etc.

2.6.1 Montaggio dei filesystem

In Linux tutte le partizioni utilizzate dal sistema sono elencate in /etc/fstab. Questo file contiene l'elenco di tutte le partizioni (o i filesystem) montate. La sintassi da utilizzare per il file /etc/fstab prevede che per ogni partizione siano specificate una serie di informazioni, raggruppate in campi ben definiti.

- Il primo campo indica la partizione, o il device, da montare.
- Il secondo campo indica il punto di montaggio della partizione.
- Il terzo campo descrive il tipo di filesystem della partizione.
- Il quarto campo elenca le opzioni che si intendono utilizzare per montare quella specifica partizione, se il filesystem sarà a sola lettura, se un utente può o non può montare la partizione ecc.
- Il quinto campo serve ad indicare se la partizione necessita dell'operazione di dump. Il valore di questo campo può essere lasciato a zero.
- L'ultimo campo è usato da fsck per determinare come i filesystem dovrebbero essere controllati nel caso in cui non siano stati smontati correttamente. Per il root filesystem questo campo dovrebbe essere impostato ad 1, per gli altri filesystem che si vuole che vengano controllati questo valore dovrebbe essere impostato a 2, mentre per quelli che non è necessario controllare, può essere lasciato a 0.

in base a tali regole, il file /etc/fstab creato è il seguente:

```
# cat fstab

# /etc/fstab: static file system information.
#
# <file system> <mount pt> <type> <options> <dump> <pass>
#
# /dev/mtdblock0 /    jffs2  defaults  1   1
#
/dev/root       /          ext2   rw     0       1
proc            /proc      proc   defaults    0       0
devpts          /dev/pts   defaults,gid=5,mode=620 0    0
tmpfs           /tmp       tmpfs  defaults    0       0
sysfs           /sys       sysfs  defaults  0         0
```

La prima partizione nell'elenco è quella contenente la distribuzione, da montare come root filesystem. Per i test il tipo del filesystem è ext2, quando la distribuzione sarà installata sulla memoria flash occorrerà modificare questo file e impostare il tipo di filesystem come jffs2. Tutti gli altri device indicano filesystem virtuali.

2.6.2 Inizializzazione del sistema

Busybox è fornito con una serie di script di inizializzazione. Per la distribuzione EFML si sono utilizzati questi script come base di partenza, modificandoli in funzione delle indicazioni presenti sul manuale della scheda e in funzione delle necessità incontrate.
Il primo file modificato è stato /etc/inittab. Si è modificato il modo in cui vengono lanciate le console.

```
::askfirst:-/bin/sh
```

è stato cambiato in

```
::respawn:/sbin/getty 38400 tty1
```

le stesse modifiche sono state apportate per la configurazione della seconda console. Per le rimanenti due si è lasciato come processo da eseguire /bin/sh, cambiando l'azione da askfirst a respawn. Gli script di sistema per l'inizializzazione sono init.d/rcS e init.d/rcL. Lo script rcS, di default si occupa di lanciare tutti gli altri script presenti in init.d, e cioè:

- S20urandom: si occupa di inizializzare e fermare il generatore di numeri casuali. Salva in /etc/random-seed il seme per la sessione corrente
- S40network: si occupa dell'inizializzazione e interruzione della rete
- S49ntp: utilizzato per far partire e fermare il demone ntp (network time protocol)
- S50sshd: inizializza e ferma secure shell. Si occupa di verificare l'esistenza dei

programmi necessari alla gestione delle chiavi e la presenza delle chiavi stesse. Nel caso in cui le chiavi non siano presenti le genera.

Oltre a ciò lo script rcS è stato modificato per montare i filesystem virtuali /proc e /sys

```
if [ ! -e /proc/mounts ]; then
mount -n -t proc /proc /proc
mount -n -t sysfs /sys /sys >/dev/null 2>&1
fi
```

stampare un messaggio di benvenuto,

```
# Wellcome banner
echo "Wellcome to Embedded Finmeccanica Linux"
```

rimontare in lettura/scrittura il root filesystem e tutti gli altri filesystem presenti in /etc/fstab.

```
# Remounting root filesystem in read-write mode
mount -n -o remount,rw /

# Mounting other filesystems
mount -a
```

Lo script rcL è stato modificato affinché settasse l'hostname e caricasse la tastiera italiana.

```
export PATH=$PATH:/bin:/sbin:/usr/bin:/usr/sbin
export LD_LIBRARY_PATH=/lib:/usr/lib
```

```
echo "Setting hostname"
/bin/hostname ewok

echo "Loading Italian keyboard..."
loadkeys /lib/kbd/keymaps/i386/qwerty/it.map.gz
```

Il file /etc/hosts contiene la traduzione tra indirizzi IP e nomi di host, utilizzato all'interno di piccole reti prive di server DNS. Il suo contenuto è stato fissato a:

```
127.0.0.1  localhost.localdomain localhost ewok
```

L'ultimo file di configurazione editato è profile. In questo file sono state esportati i percorsi delle directory contenenti le librerie condivise:

```
export LD_LIBRARY_PATH=/lib:/usr/lib
```

A termine delle varie configurazioni l'immagine della distribuzione è stata smontata ed è stata copiata sulla partizione di test creata. I comandi utilizzati per effettuare queste operazioni sono stati.

```
umount efml_dir
dd if=efml.img of=/dev/sda10 bs=1024k count=50
```

quindi si è utilizzato qemu per verificate la correttezza delle operazioni sin qui effettuate. Qemu è stato configurato per lanciare il kernel direttamente dalla partizione su cui è stato installato e per montare come root filesystem quello presente sulla partizione su cui si è copiata l'immagine della distribuzione (figura 21).

/serio1/input/input1
[2.853936] I2O subsystem v1.325
[2.854015] i2o: max drivers = 8
[2.857635] I2O Bus Adapter OSM v1.317
[2.858149] I2O Block Device OSM v1.325
[2.859965] i2c /dev entries driver
[2.870580] usbcore: registered new interface driver hiddev
[2.871174] usbcore: registered new interface driver usbhid
[2.871445] drivers/hid/usbhid/hid-core.c: v2.6:USB HID core driver
[2.872345] TCP cubic registered
[2.872485] NET: Registered protocol family 1
[2.872590] NET: Registered protocol family 17
[2.874702] Using IPI Shortcut mode
[2.886191] EXT2-fs warning: mounting unchecked fs, running e2fsck is recommended
[2.886690] VFS: Mounted root (ext2 filesystem).
[2.907976] Freeing unused kernel memory: 248k freed
Mounting proc filesystem...
Mounting sys filesystem...
Initializing random number generator... done.
Starting network...
ifup: interface lo already configured
Loading italian keyboard...
Loading lib/kbd/keymaps/i386/qwerty/it.map.gz
_

Figura 21: Prompt di EFML in qemu

2.6.3 Immagine su filesystem jffs2

Al termine dei vari test effettuati sulla macchina di sviluppo e dopo aver verificato la corretta funzionalità della distribuzione, è stato creato il root filesystem che sarà installato sulla memoria flash. La distribuzione Embedded Finmeccanica Linux è stata installata, presso i laboratori della MBDA, su di una macchina di test alla quale era collegata la scheda VP417. Il kernel della distribuzione è stato ricompilato dopo che il driver MTD è stato modificato nel modo seguente:

```
/* Size of application flash (Default value 65536 = 64MB) */
#define MAX_SIZE_KB      65536

/* Number of logical partitions to split the flash device into */
#define NUM_PARTITIONS 1

/* Partition sizing, partition sizes must be in multiples of 128K
*/
```

```
#define P1_PARTITION_SIZE_KB  65536
#define P2_PARTITION_SIZE_KB  0
#define P3_PARTITION_SIZE_KB  0
#define P4_PARTITION_SIZE_KB  0

/* Partition names, the logical partitions will register under
 * the MTD subsystem with the names defined here */
#define P1_PARTITION_NAME    "MTD flash partition"
#define P2_PARTITION_NAME    ""
#define P3_PARTITION_NAME    ""
#define P4_PARTITION_NAME    ""
```

Si è deciso di creare una sola partizione, occupante l'intera flash avente una capacità pari a 64Mb e identificata dall'etichetta “MTD flash partition”. Dopo aver riavviato il sistema si è dato il comando

```
cat /proc/mtd
```

l'output ottenuto è stato

```
dev   size       erasesize  name
mtd0  04000000   00020000   “MTD flash partition”
```

questi valori sono stati utilizzati per generare, partendo dalla directory contenente la distribuzione, l'immagine del filesystem formattato jffs2 da installare sulla memoria flash. Si è entrati nel chroot di gentoo e da qui si è generata l'immagine. I comandi utilizzati sono stati:

```
mount -t proc none /media/gentoo/proc
```

```
mount -o bind /dev /media/gentoo/dev
chroot /media/gentoo /bin/bash

mkfs.jffs2 -d flash2 -e 0x20000 -p 0x4000000 \
           -o mtdflash2.img
```

al termine si è dato il comando ls -l

```
# ls -l
totale 306M
drwxr-xr-x  2 root root 4,0K 17 set 12:23 bin
drwxr-xr-x  3 root root 4,0K 10 lug 08:35 boot
...
drwxr-xr-x 14 root root 1,0K 27 set 15:20 flash2
-rw-r--r--  1 root root 50M 16 set 14:06 flash2.img
-rw-r--r--  1 root root 54M 27 set 15:32 flash2.iso
...
-rw-r--r--  1 root root 24M 5 ott 14:59 mtdflash2.img
...
```

l'immagine mtdflash2.img ha dimensione pari a 24 Mb, per poterla installare sulla flash è necessario che questa sia prima di tutto cancellata, quindi si può procedere alla copia e al montaggio della partizione:

```
dd if=/dev/zero of=/dev/mtdblock0 bs=128k count=512[21]
dd if=mtdflash2.img of=/dev/mtdblock0

mkdir /mnt/flash
mount -t jffs2 /dev/mtdblock0 /mnt/flash
```

21 Device size /erase size (0x4000000 / 0x20000 = 0x200 = 512)

Capitolo 3

Sviluppi

In questo capitolo verrà esaminata la struttura del file system della distribuzione Embedded Finmeccanica Linux, quindi si daranno alcune metriche. Si verificherà sia quanta memoria occupa il kernel, in ram e su disco, sia a quanto ammonta lo spazio occupato dall'intero file system sul disco. Dall'osservazione dello spazio occupato dal filesystem si potrà individuare su quali directory, eventualmente intervenire per ridurre lo spazio occupato. Oltre a ciò si cercherà di esaminare i tempi impiegati dal sistema ad effettuare il boot.

3.1 Struttura del filesystem

La struttura del filesystem per i sistemi operativi unix-like, come GNU/Linux, è definita dal Filesystem Hierarchy Standard[22] (FHS). Le FHS definiscono quali sono e come sono organizzate le directory principali che formeranno il filesystem. L'obiettivo del FHS è quello di uniformare la struttura del filesystem su tutti i sistemi unix-like, quindi non solo GNU, ma anche su i vari BSD. Lo standard è attualmente gestito dalla Free Standard Group, una associazione non-profit composta dai maggiori produttori di hardware e software. Le directory che compaiono nel livello principale del filesystem sono le seguenti:

- bin: utilizzata per i programmi essenziali.
- boot: contiene i file necessari al bootloader e il kernel.
- dev: contiene i file identificatori dei device.
- etc: usato per i file di configurazione del sistema.
- home: la directory che contiene le home degli utenti.
- lib: contiene le librerie essenziali per il sistema, come le librerie C e i moduli del

22 http://www.pathname.com/fhs/

kernel.

- mnt: la directory utilizzata per montare altri filesystem al ramo principale
- opt: usata per installare software opzionale
- proc: filesystem virtuale creato dal kernel.
- root: home directory dell'amministratore del sistema.
- sbin: programmi essenziali all'amministratore.
- tmp: directory contenente file temporanei.
- usr: è una directory contenente una gerarchia di sottodirectory simile alla radice. Utilizzata per altri file binari, documentazione, librerie, ecc.
- var: contiene dati variabili generati da programmi e demoni durante la loro esecuzione.

Osservando l'organizzazione attuale del filesystem si possono notare alcune ridondanze, che insieme alcune scelte, ereditate dai vecchi Unix, possono lasciare perplessi i neofiti di questi sistemi. Alcune delle directory contenute nel livello principale del filesystem sono pensate per i sistemi multiutente, e risultano sostanzialmente inutili per i sistemi embedded. In generale un sistema embedded non ha bisogno di creare esattamente la gerarchia di directory definita nello standard. Su di una generica workstation unix, utilizzata da più utenti, è normale trovare la directory /home, al cui interno saranno create le directory degli utenti del sistema, è normale trovare la directory root, assegnata all'amministratore del sistema ed è utile che siano presenti le directory /mnt o /opt. Un sistema embedded non ha certamente bisogno della directory /home, e potrebbe tranquillamente fare a meno sia della /mnt che della /opt (utilizzata raramente anche sui sistemi desktop). La creazione di un filesystem comporta la selezione di alcune directory e di alcuni file necessari alla corretta esecuzione del sistema. Le directory che sono essenziali per il corretto funzionamento del sistema sono /bin, /dev, /etc, /lib, /proc, /sbin e /usr, più alcune altre loro sottodirectory. La corretta creazione del filesystem prevede oltre alla struttura delle directory la presenza di:

- Alcuni file eseguibili, necessari all'amministrazione del sistema stesso, come ls, cp,

sh, mv.

- Librerie di sistema, le librerie C che forniscono agli altri programmi installati una serie di servizi
- Alcuni file di configurazione, come ad esempio fstab, inittab, profile o i vari script contenuti all'interno della directory init.d
- Alcuni file speciali, i device node, necessari per accedere all'hardware presente sulla macchina.

3.2 Filesystem di EFML

Le directory create per la distribuzione Embedded Finmeccanica Linux sono state le seguenti:

/bin, /boot, /dev, /etc, /lib, /proc, /root, /sys, /sbin, /tmp, /usr, /var

Si è provveduto a creare solo la gerarchia di directory principale, tutte le sotto directory sono state create dall'esecuzione degli script di installazione dei singoli pacchetti. I file che costituiscono il kernel sono contenuti in due diverse directory, il kernel vero e proprio è copiato nella directory /boot, mentre i moduli e i driver sono contenuti in una sottodirectory di /lib/modules, avente per nome un numero identico alla versione del kernel a cui fanno riferimento. In questo modo è possibile avere installati sulla stessa macchina più versioni di kernel e moduli.

La directory /dev ha, tradizionalmente, per i sistemi unix un ruolo speciale. Contiene dei file, o dei nodi, relativi ai device presenti sulla macchina. La directory /dev non conterrà nei sistemi embedded la mole di device presenti su una macchina desktop o su di un server. I device sono dei file speciali, vanno creati dal root utilizzando il comando mknod, sono associati all'hardware presente sulla macchina e vengono utilizzati per accedere all'hardware cui fanno riferimento. Dalla pagina di manuale di mknod si ricava la sintassi da utilizzare per creare un device node:

mknod [OPTION]... NAME TYPE [MAJOR MINOR]

in base alla quale, per creare il device node di tty0, si scrive:

```
mknod -m 420 tty0 c 4 0
```

il device tty0, è un device a caratteri (c oppure u), altri device possono essere a blocchi (b) o fifo (p), il major e minor number identificano il dispositivo e devono essere indicati quando il tipo è c, u oppure b. Con lo switch m si settano i permessi per accedere al device. Un'altra directory particolare è /proc. Da questa directory è possibile recuperare tutte le informazioni vitali sullo stato del sistema. Proc non è una vera e propria directory, è un filesystem virtuale, è creato dinamicamente dal kernel e riflette lo stato sia dell'hardware che dello stesso kernel e di tutti i processi che sono stati lanciati e sono in quel momento in esecuzione.

Gli eseguibili necessari all'amministrazione del sistema sono stati installati sul filesystem mediante il pacchetto busybox.

3.2.1 Spazio occupato dal filesystem di EFML

Per determinare quanto spazio occupa sul disco il filesystem si possono utilizzare alcuni dei comandi messi a disposizione dai sistemi Gnu/Linux. I comandi in questione sono *df, du, ls*. Il comando *df* è utilizzato per esaminare quanto spazio occupa un file system sul disco, elenca tutte le partizioni presenti sui dischi, il punto di montaggio, la dimensione della partizione, lo spazio libero e lo spazio occupato. Dopo aver copiato l'immagine del filesystem di EFML su di una partizione creata appositamente ed aver dato il comando df si è ottenuto il seguente output

```
# df
Filesystem     Dimens. Usati Disp. Uso% Montato su
/dev/sda3       24G  14G 9,3G 60% /
tmpfs          1,5G  72K 1,5G  1% /dev/shm
...
```

```
/dev/sda11      49M  46M 922K 99% /media/test_flash
```

Seguendo le specifiche la partizione /media/test_flash, montata sul device /dev/sda11, su cui è stata installata la distribuzione ha una capacità di 50 Mb. Dall'esecuzione del comando df, si può rilevare che dello spazio a disposizione ne è stato occupato il 99%, pari a 46Mb. Oltre a df è possibile utilizzare il comando du, che produrrà un output più dettagliato sull'occupazione di spazio dei singoli file. Eseguendo du si otterrà una lista dei file e delle directory presenti nella partizione con relativo spazio occupato.

```
# du -h
2,0K  ./sys
4,5M  ./bin
155K  ./boot/grub
3,1M  ./boot
2,0K  ./dev/shm
3,0K  ./dev/net
...
2,0K  ./var/log
2,0K  ./var/tmp
12K   ./var
47M   .
```

L'output prodotto da *du*, stampa lo spazio occupato da ciascun file e directory, sottodirectory comprese. Una analisi di questi dati permette di determinare la directory che necessita di più spazio ed eventualmente intervenire cercando di apportare le necessarie modifiche. Un ultimo comando da usare dalla shell per ottenere dati sullo spazio occupato è *ls*, seguito dagli switch -l -p -h

```
# ls -l -p -h
totale 31K
drwxr-xr-x 2 root root 2,0K 16 set 16:08 bin/
drwxr-xr-x 3 root root 1,0K 18 set 16:34 boot/
drwxr-xr-x 6 root root 2,0K 18 set 16:33 dev/
drwxr-xr-x 5 root root 1,0K 16 set 10:48 etc/
drwxr-xr-x 5 root root 2,0K 18 set 16:34 lib/
lrwxrwxrwx 1 root root  11 16 set 16:07 linuxrc -> bin/busybox
...
drwxr-xr-x 7 root root 1,0K 16 set 10:27 usr/
drwxr-xr-x 7 root root 1,0K 16 set 13:59 var/
```

Il comando stamperà per ogni file o directory i permessi, il proprietario, il gruppo a cui appartiene il proprietario del file, la dimensione e la data di creazione del file. Le stesse informazioni possono essere recuperate utilizzando degli appositi tool grafici (baobab nel caso in esame), eseguiti sul filesystem contenente la distribuzione in oggetto.

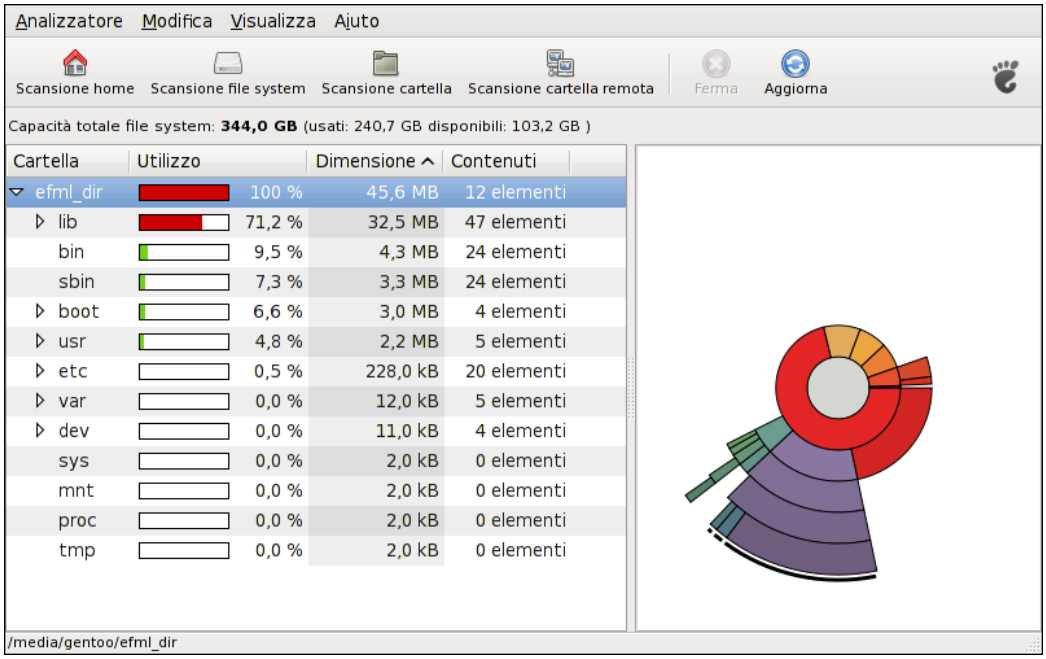

Analizzatore Modifica Visualizza Aiuto
Scansione home
Scansione file system
Scansione cartella
Scansione cartella remota
Ferma
Aggiorna
Capacità totale file system: 344,0 GB (usati: 240,7 GB disponibili: 103,2 GB)
Cartella Utilizzo Dimensione Contenuti
efml_dir 100 % 45,6 MB 12 elementi
lib 71,2 % 32,5 MB 47 elementi
bin 9,5 % 4,3 MB 24 elementi
sbin 7,3 % 3,3 MB 24 elementi
boot 6,6 % 3,0 MB 4 elementi
usr 4,8 % 2,2 MB 5 elementi
etc 0,5 % 228,0 kB 20 elementi
var 0,0 % 12,0 kB 5 elementi
dev 0,0 % 11,0 kB 4 elementi
sys 0,0 % 2,0 kB 0 elementi
mnt 0,0 % 2,0 kB 0 elementi
proc 0,0 % 2,0 kB 0 elementi
tmp 0,0 % 2,0 kB 0 elementi
/media/gentoo/efml_dir

Figura 22: Baobab - Spazio occupato dal filesystem di EFML

Anche visivamente è possibile osservare che l'intera distribuzione occupa un totale di 45.6 Mb (figura 22). La maggior parte di questo spazio sul filesystem, il 71.2% equivalente a 32.5 Mb è occupato dalla directory /lib.

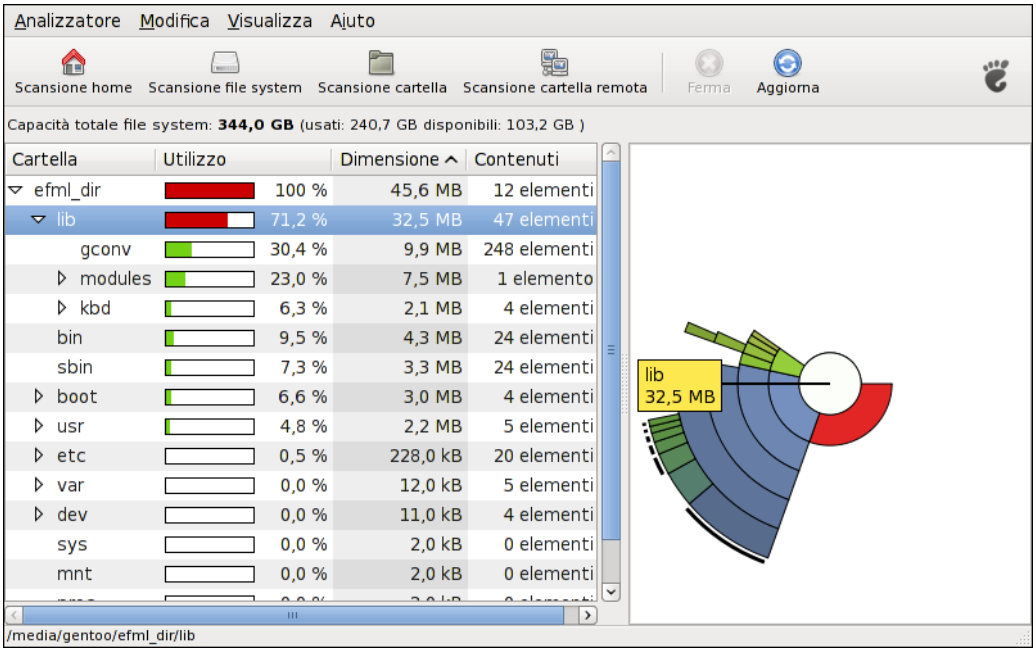

Analizzatore Modifica Visualizza Aiuto
Scansione home
Scansione file system
Scansione cartella
Scansione cartella remota
Ferma
Aggiorna
Capacità totale file system: 344,0 GB (usati: 240,7 GB disponibili: 103,2 GB)
Cartella Utilizzo Dimensione Contenuti
efml_dir 100 % 45,6 MB 12 elementi
lib 71,2 % 32,5 MB 47 elementi
gconv 30,4 % 9,9 MB 248 elementi
modules 23,0 % 7,5 MB 1 elemento
kbd 6,3 % 2,1 MB 4 elementi
bin 9,5 % 4,3 MB 24 elementi
sbin 7,3 % 3,3 MB 24 elementi
boot 6,6 % 3,0 MB 4 elementi
usr 4,8 % 2,2 MB 5 elementi
etc 0,5 % 228,0 kB 20 elementi
var 0,0 % 12,0 kB 5 elementi
dev 0,0 % 11,0 kB 4 elementi
sys 0,0 % 2,0 kB 0 elementi
mnt 0,0 % 2,0 kB 0 elementi
lib
32,5 MB
/media/gentoo/efml_dir/lib

Figura 23: Baobab - Spazio occupato dalle sotto directory di /lib

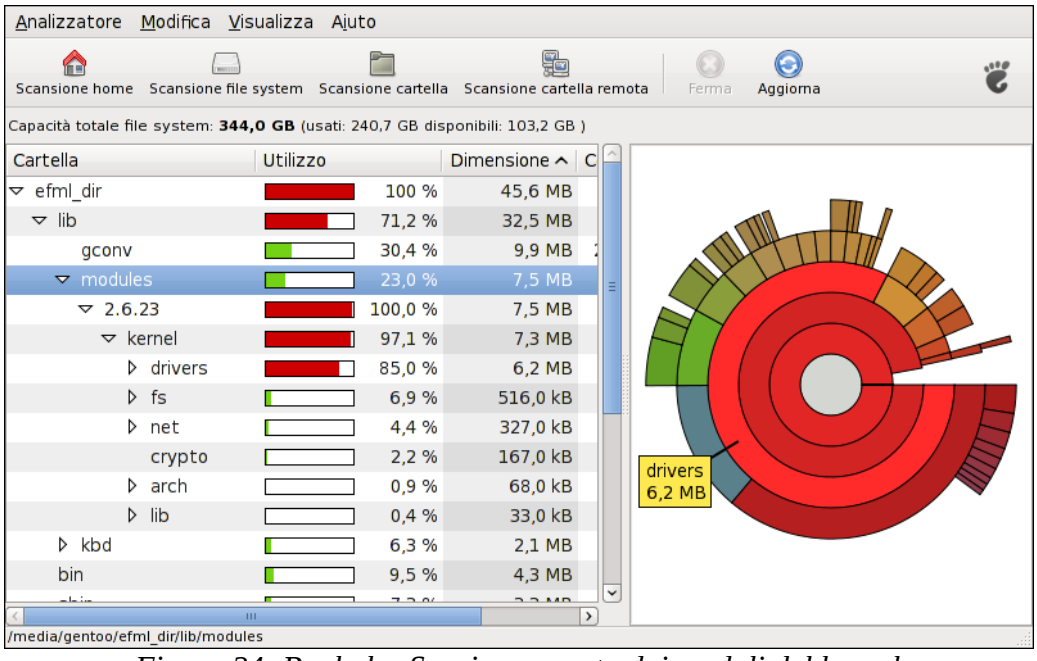

Analizzatore Modifica Visualizza Aiuto
Scansione home
Scansione file system
Scansione cartella
Scansione cartella remota
Ferma
Aggiorna
Capacità totale file system: 344,0 GB (usati: 240,7 GB disponibili: 103,2 GB)
Cartella Utilizzo Dimensione
efml_dir 100 % 45,6 MB
lib 71,2 % 32,5 MB
gconv 30,4 % 9,9 MB
modules 23,0 % 7,5 MB
2.6.23 100,0 % 7,5 MB
kernel 97,1 % 7,3 MB
drivers 85,0 % 6,2 MB
fs 6,9 % 516,0 kB
net 4,4 % 327,0 kB
crypto 2,2 % 167,0 kB
arch 0,9 % 68,0 kB
lib 0,4 % 33,0 kB
kbd 6,3 % 2,1 MB
bin 9,5 % 4,3 MB
drivers
6,2 MB
/media/gentoo/efml_dir/lib/modules

Figura 24: Baobab - Spazio occupato dai moduli del kernel

Di questo, il 30.4% è occupato dalla sotto directory gconv per un totale di circa 10 Mb,

mentre un altro 6.3%, pari a 2.1 Mb, è occupato dalla directory contenente i file e le mappe necessaire alla gestione delle varie tastiere. L'altra sotto directory, modules, contenente i driver del kernel, occupa complessivamente circa 7.5 Mb pari al 23%. La directory gconv è parte della glibc, si può quindi concludere che circa 23 Mb dei 32.5 Mb occupati dalla directory /lib sono attribuibili alla libreria glibc (figura 23). Questo risultato era comunque atteso, visto che per i sistemi embedded è molto diffuso l'uso di librerie alternative alla glibc, come la uClibc. Della sotto directory contenente i moduli del kernel (figura 24), una parte considerevole dello spazio è occupato dai driver dei dispositivi hardware, seguiti dai moduli di rete. Per poter ulteriormente ridurre l'occupazione è necessario intervenire sulla configurazione del kernel, abilitando i driver per il solo l'hardware da gestire.

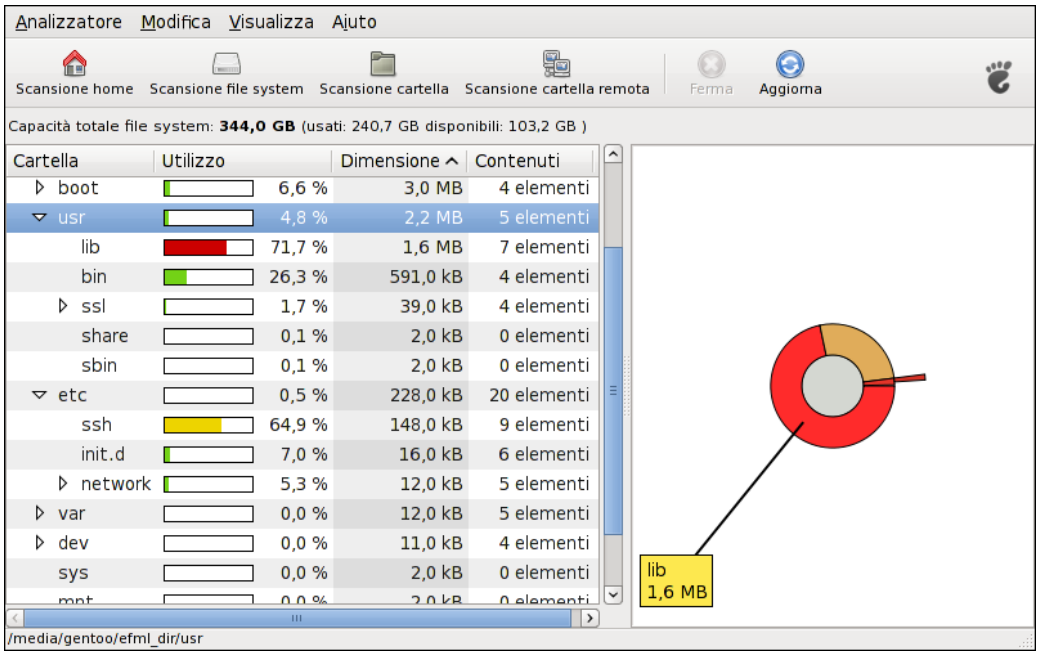

Analizzatore Modifica Visualizza Aiuto
Scansione home
Scansione file system
Scansione cartella
Scansione cartella remota
Ferma
Aggiorna
Capacità totale file system: 344,0 GB (usati: 240,7 GB disponibili: 103,2 GB)
Cartella Utilizzo Dimensione Contenuti
boot 6,6 % 3,0 MB 4 elementi
usr 4,8 % 2,2 MB 5 elementi
lib 71,7 % 1,6 MB 7 elementi
bin 26,3 % 591,0 kB 4 elementi
ssl 1,7 % 39,0 kB 4 elementi
share 0,1 % 2,0 kB 0 elementi
sbin 0,1 % 2,0 kB 0 elementi
etc 0,5 % 228,0 kB 20 elementi
ssh 64,9 % 148,0 kB 9 elementi
init.d 7,0 % 16,0 kB 6 elementi
network 5,3 % 12,0 kB 5 elementi
var 0,0 % 12,0 kB 5 elementi
dev 0,0 % 11,0 kB 4 elementi
sys 0,0 % 2,0 kB 0 elementi
lib
1,6 MB
/media/gentoo/efml_dir/usr

Figura 25: Baobab - Spazio occupato dalla directory /usr

Delle rimanenti directory presenti, si può osservare che:

- una parte consistente dello spazio complessivo, 4.3 Mb pari al 9.5% è utilizzato dalla directory /bin (figura 22).
- il kernel e i file necessari al bootloader occupano 3Mb pari al 6.6% del disco (figura 22)

- Le librerie occupano, ancora, una quantità discreta di spazio su disco (/usr/lib) (figura 25)
- I file di configurazione in /etc e gli eseguibili in /usr/bin occupano uno spazio mimino (figura 25)

Tuttavia, è opportuno ricordare che la distribuzione realizzata presenta diversi software non richiesti dalle specifiche, come le kbd per la gestione delle tastiere diverse da quella statunitense e le mtd utils, con la relativa libreria lzo, per una più semplice amministrazione della memoria a stato solido.

3.3 Occupazione di memoria del kernel

Il kernel Linux, diversamente da tutti gli altri programmi, non è sottoposto a paginazione, ciò significa che il kernel (testo, stack e dati) risiede sempre in memoria e non è mai sottoposto a swapping. É importante, particolarmente per i sistemi embedded, capire come stimare l'occupazione di memoria del kernel e identificare alcuni modi per, eventualmente, ridurla. La prima verifica che si può fare è relativa allo spazio occupato sul disco.

```
# ls -l -h vmlinuz-2.6.23
-rwxr-xr-x 1 root root 2,0M 23 set 10:00 vmlinuz-2.6.23
```

Il kernel occupa 2 Mb di memoria. Altre informazioni sull'occupazione del kernel in ram si possono ottenere esaminando alcune informazioni messe a disposizione dallo stesso kernel. Si esamina l'output di dmesg:

```
# dmesg
...
Real-Time Preemption Support (C) 2004-2007 Ingo Molnar
Built 1 zonelists in Zone order. Total pages: 226240
Kernel command line: rw root=/dev/sda10 vga=0x317
...
```

```
Memory:  899368k/917504k  available  (2576k  kernel  code,
       17636k reserved, 1012k data, 216k init, 0k highmem)
virtual kernel memory layout:
     fixmap : 0xfffb5000 - 0xfffff000  ( 296 kB)
     vmalloc : 0xf8800000 - 0xfffb3000  ( 119 MB)
     lowmem : 0xc0000000 - 0xf8000000  ( 896 MB)
     .init : 0xc0788000 - 0xc07be000  ( 216 kB)
     .data : 0xc0684342 - 0xc07815dc  (1012 kB)
     .text : 0xc0400000 - 0xc0684342  (2576 kB)
...
squashfs: version 3.2-r2 (2007/01/15) Phillip Lougher
JFFS2 version 2.2. (NAND) (SUMMARY) © 2001-2006 Red Hat, Inc.
```

in memoria circa 2,5 Mb sono occupati dal testo del kernel, circa 17 Mb sono riservati, un mega è riservato alle strutture dati.

Il comando free restituisce alcune interessanti informazioni. Dà in output la quantità di memoria fisica occupata e disponibile sul sistema, quanto swap è usata (informazione che tuttavia per il sistema embedded in costruzione non è utilizzata dato che la swap non è stata attivata, sia per le limitate risorse disponibili sulla flash, sia per non aumentare il carico sul filesystem jffs2 con ripetute operazioni di lettura/scrittura e conseguente abbassamento della vita media dei blocchi della memoria a stato solido), quanto spazio è occupato dai buffer del kernel.

```
# free
   total   used   free  shared  buffers
Mem:  899696   18560  881136    0    340
Swap:    0     0     0
Total: 899696   18560  881136
```

La memoria totale disponibile sul sistema è di circa 900 Mb, quella utilizzata è di 18Mb, non c'è memoria condivisa, i buffer del kernel utilizzano 340 Kb. Come ci si attendeva, la swap, non essendo abilitata, non occupa spazio.

La memoria occupata a runtime è esportata dal kernel sotto il file meminfo del filesystem virtuale proc:

```
# cat /proc/meminfo
MemTotal:    899696 kB
MemFree:     881508 kB
Buffers:       332 kB
Cached:       6684 kB
SwapCached:      0 kB
Active:       2724 kB
Inactive:     4632 kB
SwapTotal:       0 kB
SwapFree:        0 kB
Dirty:          16 kB
Writeback:       0 kB
AnonPages:     340 kB
Mapped:       412 kB
Slab:        3004 kB
SReclaimable:   580 kB
SUnreclaim:    2424 kB
PageTables:     60 kB
NFS_Unstable:    0 kB
Bounce:          0 kB
CommitLimit:  449848 kB
```

```
Committed_AS:   2440 kB
VmallocTotal:  122572 kB
VmallocUsed:   3648 kB
VmallocChunk:  118472 kB
HugePages_Total:   0
HugePages_Free:   0
HugePages_Rsvd:   0
Hugepagesize:   4096 kB
```

Esistono varie tecniche per ridurre l'occupazione di memoria da parte del kernel. Si può intervenire sulle strutture dati allocate staticamente. Molte strutture dati non hanno una opzione di configurazione, la quantità di strutture che il kernel creerà è definita in modo statico, il che può portare ad un cattivo uso della memoria. Ad esempio nel file include/linux/tty.h è definito il numero di console tty da creare di default, il loro valore è fissato attraverso le variabili MAX_NR_CONSOLES e MAX_NR_USER_CONSOLES, i cui valori sono settati a 63. Per un sistema embedded questi valori potrebbero essere tranquillamente ridotti di molto. Nel file kernel/printk.c si setta LOG_BUF_LEN, la grandezza del buffer di log, pari a 16K. Ancora, nel file include/linux/major.h si può definire il numero massimo di device a blocchi e a caratteri che è possibile creare sotto la directory /dev, cambiando i valori impostati per MAX_CHRDEV e MAX_BLKDEV. Tutte queste ottimizzazioni si effettuano partendo dal presupposto che un sistema embedded non necessita di tutte le risorse disponibili su un sistema desktop o server. La prima ottimizzazione che si può effettuare è, naturalmente, in fase di configurazione, eliminando tutto il codice che gestisce l'hardware non presente sul sistema embedded. Una ulteriore ottimizzazione può essere effettuata sul modo in cui si alloca la memoria dinamica. Il kernel, attraverso la funzione kmalloc(), fornisce ai programmi che ne fanno richiesta blocchi di memoria allocata dinamicamente. Questi blocchi di memoria sono multipli di 32 bytes, il che può portare a frammentazione della memoria. Se ad esempio un

programma ha bisogno di 80 bytes, la quantità che gli verrà allocata sarà un multiplo di 32 bytes, quindi 96 bytes, con una frammentazione di 16 bytes. Se il programma in oggetto effettuasse molte richieste di allocazione di memoria, allora la quantità di ram inutilizzata diventerebbe presto molto alta. Una possibile soluzione potrebbe essere la creazione di una apposita funzione che alla partenze del programma chiede che le venga allocata una congrua quantità di memoria e poi la gestisca internamente nel modo più appropriato.

3.4 Tempi di boot

Uno dei campi di maggiore sviluppo degli ultimi anni è relativo ai tempi di boot dei sistemi basati su kernel Linux e inizializzazione basata sul SysV init. Il tempo di boot è composto da diversi contributi. Dal momento in cui la macchina è accesa al momento in cui viene eseguito il primo programma utente, bisogna considerare quanto tempo impiega il bios ad effettuare le operazioni di POST e ad inizializzare l'hardware, quanto tempo è necessario a caricare il bootloader e quanto tempo il kernel impiega ad inizializzare tutti i suoi sottosistemi e lanciare l'init, l'ultimo contributo è dovuto al tempo impiegato da init ad eseguire i vari servizi.

3.4.1 TSC - Time Stamp Clock

Non è semplice misurare i singoli contributi che concorrono alla creazione della latenza del boot. I processori Intel, dal pentium in poi, dispongono di un registro chiamato TSC (Time Stamp Clock) nel quale sono registrati il numero di ticks trascorsi dalla partenza del processore. Il registro TSC fornisce un tempo molto accurato, sempre che la frequenza di clock non cambi durante la fase di boot, cosa che normalmente non accade. Durante la fase di configurazione del kernel si è abilitata l'opzione “show timing information on printk”, presente nel sotto menu “Kernel hacking”. Abilitando questa opzione, durante la fase di boot, ogni messaggio prodotto dal kernel viene preceduto da una stringa contenente il valore del registro TSC convertito in secondi. Ciò permette di avere una stima sufficientemente accurata del tempo impiegato dal kernel e dai suoi sottosistemi per

completare le proprie operazioni. Da questa misura sono tuttavia esclusi sia il tempo impiegato dal bios che dal bootloader e dall'init. Per poter avere una stima di questi ritardi sarebbe necessario esaminare il valore del registro TSC. Questa operazione può essere effettuata utilizzando il seguente programma:

```
#include <stdint.h>
#include <stdio.h>

inline uint64_t rdtsc() {
     uint32_t lo, hi;
     /* cpuid will serialize the following rdtsc with
       respect to all other instructions the processor
       may be handling
     */
     __asm__ __volatile__ (
          “xorl %%eax, %%eax\n”
          “cpuid\n”
          “rdtsc\n”
     : “=a” (lo), “=d” (hi)
     :
     : “%ebx”, “%ecx”);
return (uint64_t) hi << 32 | lo;
}

int main (void) {
     printf(“TSC: %lld\n”, rdtsc());
t1:
     goto t1;
```

```
return 0;
}
```

Questo sorgente va compilato con nel modo seguente:

```
gcc timestamp.c -static -o timestamp
```

Una volta copiato nel root filesystem l'eseguibile, questo viene caricato al boot aggiungendo il comando init=/sbin/timestamp all'elenco dei parametri passati al kernel.
Per l'analisi dei ritardi introdotti al boot ci si è limitati ai soli dati forniti dallo stesso kernel attraverso il Time Stamp Clock. Dopo aver effettuato il reboot della macchina di sviluppo, si è provveduto a redirigere l'output di dmesg in un apposito file di log. Dall'esame di questo file si sono potuti ricavare i tempi di inizio dell'esecuzione del kernel e il tempo in cui è stato lanciato il processo init.

```
# dmesg > dmesg.log
# cat dmesg.log

0000000 0008e3bd 00000000 00000001 00000001
[  17.271568] monitor/mwait feature present.
[  17.271570] CPU: L1 I cache: 32K, L1 D cache: 32K
[  17.271572] CPU: L2 cache: 3072K
[  17.271573] CPU: Physical Processor ID: 0
[  17.271574] CPU: Processor Core ID: 1
...
[  23.079134] TCP cubic registered
[  23.088044] NET: Registered protocol family 1
[  23.096906] NET: Registered protocol family 17
[  23.106059] Using IPI Shortcut mode
```

```
[  23.951558] input: ImPS/2 Synaptics TouchPad as
          /devices/platform/i8042/serio1/input/input2
[  24.022776] VFS: Mounted root (ext2 filesystem).
[  24.031704] Freeing unused kernel memory: 240k freed
```

il kernel è partito dopo circa 17 secondi e 271 centesimi ed ha liberato la memoria dopo circa 24 secondi e 31 centesimi. Bisogna osservare che questi tempi comprendono anche i ritardi introdotti dall'operatore nel selezionare l'opportuna voce del menu di grub per caricare il kernel desiderato. Da questi dati risulta che il kernel è stato caricato in circa 6,76 secondi.

3.4.2 Bootchart

Un altro sistema per osservare la latenza del boot è quella di utilizzare dei tool appositamente studiati allo scopo, uno di questi è bootchart[23]. Bootchart colleziona le informazioni riguardo all'utilizzazione delle risorse durante il boot, e permette, in seguito, di rappresentare questi dati sotto forma di grafico. Anche bootchart necessita una modifica alla normale procedura di boot del sistema, il primo processo ad essere mandato in esecuzione dal kernel non è init, bensì il demone bootchartd, che una volta avviato provvederà ad eseguire init per completare la normale procedura di boot. Questo demone sarà eseguito in background e durante l'esecuzione di init collezionerà varie statistiche prelevando i dati dal filesystem /proc. Quando init avrà terminato di eseguire l'inizializzazione del sistema, le statistiche raccolte saranno salvate nel file /var/log/bootchart.tgz. Questo file potrà poi essere processato da una applicazione Java che creerà un grafico contenente l'albero dei processi chiamati. Analizzando questo grafico si potranno determinare le dipendenze tra i processi e le risorse da loro utilizzate. Per la distribuzione Embedded FinMeccanica Linux, l'uso di bootchart non è tuttavia giustificato, vista le esigue operazioni che init compie dopo il montaggio del filesystem /proc. [45]

23 http://www.bootchart.org

3.4.3 Upstart

Si è osservato che i tempi di boot potrebbero essere notevolmente accorciati se si apportassero delle modifiche al sistema di inizializzazione. Init, per caricare tutti i demoni e i servizi, utilizza un procedimento sequenziale. Quando init lancia un processo, prima di procedere al lancio del successivo, attende che quello corrente sia ritornato. Questa tecnica può essere utile se il processo successivo ha la necessità di utilizzare un qualche servizio fornito dal processo (demone) che in quel momento sta eseguendo, cioè se esiste una qualche dipendenza tra i vari processi da lanciare. Tuttavia questi sono casi isolati e la maggior parte dei servizi sono tra loro indipendenti. Da questa osservazione si stanno sviluppando nuove tecniche per poter lanciare i vari demoni, non più in modo sequenziale, bensì in parallelo, ottenendo un notevole risparmio sui tempi di boot. La Canonical Ltd[24], realizzatrice della distribuzione Linux Ubuntu, ha sostituito lo script di avvio init con il più moderno Upstart[25]. Questo demone garantisce la compatibilità con l'init SysV e con i suoi script, quindi come il classico init è utilizzato per le operazioni di start, stop e restart dei vari processi. Utilizza gli script di configurazione già esistenti, ma a differenza di init, agisce simultaneamente su più servizi e processi. La mancanza dello script init ha come conseguenza, su quelle distribuzioni che utilizzano upstart, la mancanza del file di configurazione /etc/inittab. Attualmente upstart è utilizzato sulle distribuzioni Ubuntu, è da poco disponibile per le distribuzioni Fedora, ed è in fase di test sulle Debian.

[40], [41], [44]

24 http://www.canonical.com
25 http://upstart.ubuntu.com

Capitolo 4

Test

"Lo sviluppo di sistemi software comporta una serie di attività produttive nelle quali le occasioni di inserire errori umani abbondano. Gli errori possono verificarsi fin dal principio del processo, quando gli obiettivi possono essere specificati in modo errato o imperfetto, fino alle ultime fasi di progettazione e sviluppo. A causa dell'incapacità umana di operare e comunicare in modo perfetto, lo sviluppo di software è affiancato da un'attività di garanzia di qualità" (Deutsch, 1979). Dopo aver creato un software è necessario che questo sia collaudato, di modo che siano scoperti e corretti il maggior numero possibile di errori, prima di fornire il prodotto al cliente. Per poter far ciò è necessario pianificare una serie di test atti ad individuare gli eventuali errori. L'ingegneria del software definisce due metodi per effettuare il collaudo dei programmi: i collaudi white-box e i collaudi blak-box. I primi sfruttano la struttura del codice da testare per creare i casi di prova, in modo da garantire che i vari cammini all'interno di un modulo siano eseguiti almeno una volta, che i rami vero e falso di una decisione siano entrambi eseguiti, ecc. I secondi si concentrano, invece, sui requisiti funzionali del software. Si cerca, cioè, di ricavare un insieme di condizioni di ingresso che esercitino i requisiti funzionali del programma. I test black-box tentano di rilevare funzioni errate o mancanti, errori nelle strutture dati, errori di comportamento o prestazionali, errori di inizializzazione e terminazione [49]. La crescente importanza del kernel Linux per le applicazioni industriali, unita ad una sua sempre più vasta diffusione in svariati ambienti, architetture e piattaforme, ha reso necessario lo sviluppo di una serie tool atti a verificarne la bontà. Si è reso necessario verificare e assicurare che linux sia un kernel sicuro, stabile e robusto e per poter garantire queste proprietà è necessario, come per ogni altro software, effettuare una serie di collaudi. I test dovranno essere eseguiti sia sul kernel che sull'intero sistema costruito attorno ad esso. Ignorare la fase di test nello sviluppo di un sistema

software complesso può portare alla produzione di un applicativo bacato, soggetto a crash frequenti, bisognoso di continui aggiornamenti, il tutto con un conseguente aumento dei costi e l'insoddisfazione dell'utente/cliente. Il modo più semplice di testare se il kernel linux e il sistema costruito su di esso funzionino, è quello di far partire la macchina e utilizzare il nuovo sistema operativo. Nonostante la sua semplicità, è comunque un test importante perché permette di verificare velocemente se quelle caratteristiche del sistema che si utilizzano maggiormente funzionano in modo corretto o meno. Il principale difetto di questo metodo è che non copre tutte le funzionalità del sistema, ma si limita alle caratteristiche e ai software maggiormente utilizzati. Il modo per coprire molte, se non tutte, delle funzionalità è quello di utilizzare delle test suite. Le test suite sono dei software scritti con il preciso scopo di collaudare e coprire il più possibile le funzionalità del sistema.

4.1 Gcov

Una delle metriche da utilizzare per misurare l'efficacia dei test è la copertura del codice. Uno strumento disponibile per verificare la copertura del codice che è stato compilato con il compilatore gcc, è gcov. Il suo utilizzo permette di produrre codice più efficiente e più veloce andando alla ricerca delle parti del programma che sono poco eseguite o che impiegano molto tempo per completare l'elaborazione. Sotto questa ottica gcov è utilizzabile come un profiler, per cercare e ottimizzare i colli di bottiglia. Dalla sua esecuzione è possibile ottenere una serie di statistiche relativamente a:

- quanto spesso le linee di codice sono eseguite
- quali linee di codice sono in esecuzione
- quanto tempo richiedono e quanta memoria consumano le varie sezioni durante la loro computazione

e dall'analisi di questi dati è possibile risalire ai moduli che è necessario ottimizzare o eventualmente riscrivere cambiando l'algoritmo implementato. Un test copre una linea di codice se esegue quella linea. L' analisi di copertura misura quanto del codice sottoposto a

test è eseguito durante i collaudi ed è un utile meccanismo per valutare la loro efficacia. Quindi, l'analisi di copertura permette di determinare la percentuale del codice di un'applicazione che è eseguito durante il test. Il più semplice test di copertura è lo "statement coverage analysis". Con questo test si suddivide il programma in blocchi, ogni blocco è la sezione di codice tra due rami. Se una istruzione del blocco è stata eseguita, allora tutto il blocco è stato coperto.

```
int *ptr = NULL;
if (condition)
     {
          ptr = &i;
          /* */
     }
*ptr = j * 10;
```

Se ogni linea dell'esempio precedente è eseguita allora la copertura è del 100%, tuttavia se la condizione è falsa il blocco di codice del ramo if non sarà eseguito e ptr non sarà settato. Quindi se la condizione è sempre falsa si dereferenzierà un puntatore a NULL esponendo il programma ad un bug. I test di copertura sono utilizzati congiuntamente alle test suite. Le test suite permettono di verificare che il comportamento del programma sia quello atteso, mentre i test di copertura permettono di verificare quanto del programma sottoposto a test dalla suite sia effettivamente testato. Le operazioni di profiling sono importanti, soprattutto per i sistemi embedded, che disponendo di poche risorse devono sfruttarle in modo ottimale. [47], [48]

4.2 Linux Test Project

Una delle suite più diffuse in ambiente Linux è il Linux Test Project (LTP), è stato creato dalla SGI[26] ed ha in seguito ricevuto contributi dalle principali aziende che ruotano attorno

26 Silicon Graphics Inc.

all'open source e al software libero come IBM, Red Hat, SuSe. La suite ad oggi, offre più di 3000 test da utilizzare per poter verificare la sicurezza la robustezza e la stabilità del kernel. I test di un software possono essere raggruppati in alcune categorie:

- Test di compilazione
- unit testing
- test funzionale
- test di sistema
- stress testing
- test di performance

Non tutti considerano la compilazione come un test, tuttavia bisogna considerare la natura multipiattaforma di Linux. Linux è compilabile su diverse architetture ed è inoltre possibile abilitare e disabilitare molte delle proprie features. Il successo della compilazione sulle varie piattaforme e architetture diventa quindi un aspetto importante nel test del kernel Linux. LTP permette di effettuare test funzionali, di sistema e di stress, mentre non offre nessun metodo per effettuare test di compilazione, performance e unit. Con la LTP si possono testare diversi aspetti del kernel, dalle system call alla gestione della memoria, dall'ipc all'I/O, dai driver al networking, dai filesystem al real time. Molti sviluppatori del kernel effettuano il suo collaudo semplicemente eseguendolo sui loro sistemi. Il test è effettuato osservando il comportamento delle applicazioni che girano sul kernel. LTP, invece, mette a disposizione degli sviluppatori un insieme di test case per verificare, attraverso l'immissione di parametri validi e non validi, il comportamento dei singoli sottosistemi che compongono il kernel. [46], [47], [48]

4.2.1 Setup della suite LTP

Per eseguite i test messi a disposizione dal Linux Test Project, sono state eseguite le seguenti operazioni:

- è stata creata una nuova partizione della dimensione di circa 2 Gb, sulla quale è stata copiata l'immagine della distribuzione Embedded Finmeccanica Linux, e

montata sulla directory /media/LTP. Durante l'esecuzione dei primi test, in base agli errori ricevuti, è stato necessario ampliare la partizione di test alla dimensione finale di 2 Gb e copiare una serie di librerie ed eseguibili. È stato necessario ricompilare busybox per inserire il supporto ad awk, tra i principali programmi richiesti affinché i test potessero essere eseguiti si ricordano python e java.

- Si è provveduto a scaricare dal sito http://ltp.sourceforge.net/, il pacchetto software ltp-full-20080831.tgz[27] contenente la suite di test. I sorgenti sono stati scompattati in /media/LTP/usr/src. Per comodità si è creato un link simbolico ltp alla directory ltp-full-20080831.
- Per evitare di dover ricreare una completa toolchain (gcc, linker, make, autotools) nella partizione di test, i sorgenti della suite sono stati compilati montando la partizione LTP sul filesystem della macchina di sviluppo e da qui installati. Per default gli eseguibili e le librerie componenti la suite di test sono installati sotto la directory /opt/ltp. Dopo l'installazione la directory /opt/ltp è stata copiata su /media/LTP.
- La macchina di sviluppo è stata riavviata ed è stata fatta partire caricando il kernel dalla nuova partizione LTP.

La directory /usr/src/ltp-full-20080831 (/usr/src/ltp) contiene, oltre ai sorgenti, anche diversi script da eseguire per effettuare varie tipologie di test. Alcuni degli script disponibili nella directory principale di ltp sono:

- IDchech.sh: per poter eseguire la test suite è necessario che sul sistema siano presenti sia il gruppo che l'utente *nobody*.
- Runltp (Runalltest.sh): è lo script che lancia la maggior parte dei test. Di default i test sono eseguiti su filesystem, I/O su disco, gestione della memoria, ipc, scheduler, system call.
- Networktest.sh: è lo script che testa la rete. Per poter eseguire questi test è necessario creare una test suite identica su una macchina server.

27 http://ltp.sourceforge.net/

- Diskio.sh: lancia i test su floppy e cdrom.
- Nella sottodirectory ltp/testcases/realtime è presente lo script run.sh, che consente di automatizzare la verifica delle caratteristiche real time del sistema. I test sul realtime possono essere eseguiti oltre che lanciando lo script ltp/testcases/realtime/run.sh anche eseguendo lo script ltp/testscripts/test_realtime.sh. Entrambi questi script richiedono un parametro che indica il tipo di test da effettuare.

Dei vari script presenti nella directory principale di ltp, quello di maggior interesse è runltp. Se lo script è lanciato senza argomenti, sarà eseguito utilizzando una serie di valori di default e cioè:

- ogni test appartenente al gruppo di default sarà eseguito una volta
- sarà prodotto un output prolisso sullo stato del test
- non sarà creato nessun file di log per il test

I test possono essere eseguiti utilizzando le impostazioni di default fornite con la suite, oppure è possibile creare un proprio insieme di test. Se si sceglie di comporre un proprio insieme di test, bisogna creare un opportuno file in cui si descrive quali programmi di test eseguire e quali parametri fornire. I programmi di test possono essere eseguiti singolarmente, richiamando l'opportuno eseguibile, con o senza parametri, oppure è possibile creare un file con l'elenco dei test da effettuare. La Linux Test Project suite contiene più di 3000 test. Di tutti questi test ne è stato selezionato per l'esecuzione solo un sottoinsieme. Come richiesto dalla documentazione allegata alla suite, si è provveduto ad eseguire preliminarmente lo script Idchech.sh, per verificare se il sistema disponesse degli utenti e dei gruppi opportuni all'esecuzione dei test, poi si è lanciato lo script ver_linux. Gli output dei due script sono i seguenti:

IDchech.sh

```
Checking for required user/group ids


'nobody' user id and group found.
```

```
'bin' user id and group found.
'daemon' user id and group found.
Users group found.
Sys group found.

Required users/groups exist.
```

ver_linux

```
...
/etc/issue:
Embedded Finmeccanica Linux
Linux ewok 2.6.23 #3 SMP PREEMPT RT Thu Sep 25 16:56:45
Local time zone must be se i686 unknown
...
mount         rootfs on / type rootfs (rw)
mount         /dev/root on / type ext2 (rw)
mount         /proc on /proc type proc (rw)
mount         /sys on /sys type sysfs (rw)
mount         devpts on /dev/pts type devpts (rw)
mount         tmpfs on /tmp type tmpfs (rw)
...
Modules Loaded

free reports:
       total   used   free  shared  buffers
Mem:  3081788   24220  3057568     0    692
Swap:    0     0     0
Total: 3081788   24220  3057568
```

```
/proc/cpuinfo
processor  : 0
vendor_id  : GenuineIntel
cpu family : 6
model      : 23
model name : Intel(R) Core(TM)2 Duo CPU   T8100 @ 2.10GHz
stepping   : 6
cpu MHz    : 2100.000
cache size : 3072 KB
physical id: 0
siblings   : 2
core id    : 0
cpu cores  : 2
...
processor  : 1
vendor_id  : GenuineIntel
...
```

4.2.2 Test delle caratteristiche real time del kernel

I primi test effettuati sono stati quelli relativi al real time, questi test case sono mantenuti dal real-time team di IBM. Per eseguire i test sul real time ci si è portati nella directory principale del Linux Test Project e si è lanciato lo script test_realtime.sh presente sotto la directory testscripts. Di seguito sono descritti sia i test eseguiti che l'output prodotto dalla loro esecuzione. Lo script è stato lanciato con il comando

```
testscripts/test_realtime.sh -t all
```

Non avendo indicato nessun insieme di test, sono stati eseguiti quelli impostati di default, definiti nel file testcases/realtime/profiles/default riportato di seguito.

```
# Test run commands for default profile
#
# format:
#    reldir testexec [ args ... ]
#
# First field is the relative directory of the test.
# Second field is the executable itself.
# Others field are arguments of the command.
# In the above example, the following would be done:
# cd reldir ; ./testexec args ...
# as a usual shell command.
#
# Comments are shell-like.
#
# This is to be read by scripts/run_c_files.sh and is useful
# for local or global runs (<RT_TESTS_ROOT>/run.sh or
# <TESTDIR>/run_auto.sh)
#

# Pass if maximum time for signaled thread to be
# scheduled is less
# than threshold (us).
# Default threshold=100 us
func/async_handler          async_handler -c 100
func/async_handler          async_handler_jk -c 100
```

```
# Pass if maximum lock time is less than threshold (us).
# Default threshold=200 us
func/pi_perf               pi_perf -c 200

# Pass if maximum latency is less than criterium (us).
# Default=20 us
func/pthread_kill_latency  pthread_kill_latency -c 20

# Pass if all treads get preempted within max loops.
# Default max=1
func/prio-preempt          prio-preempt -c 1

# Pass if all delay are less than maxduration (us).
# Default maxduration=100 us
func/sched_latency         sched_latency   -d 1 -t 5 -c 100

# Pass if ratio * average concurrent time < average
# sequential time
# Default ratio=0.75
func/matrix_mult      matrix_mult -c 0.75

# Pass if difference between the sum of thread times and
# process time
# is less than maxduration (s).
# Default maxduration=0.5 s
func/thread_clock          tc-2 -c 0.5

# The below tests have no pass/fail criterium.
```

```
func/gtod_latency          gtod_latency
func/sched_jitter          sched_jitter
func/periodic_cpu_load     periodic_cpu_load
func/periodic_cpu_load     periodic_cpu_load_single
func/prio-wake             prio-wake
func/sched_football        sched_football
func/pi-tests              testpi-0
func/pi-tests              testpi-1
func/pi-tests              testpi-2
func/pi-tests              testpi-4
func/pi-tests              testpi-5
func/pi-tests              testpi-6
func/pi-tests              sbrk_mutex
```

async_handler

Misura la latenza negli event handler asincroni. In particolare misura la latenza della chiamata pthread_cond_signal finchè il thread non è schedulato.

output `async_handler-c100.log`

```
--- Running testcase async_handler -c 100 ---
Fri Oct 3 13:09:21 UTC 2008
Logging      to      /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-async_handler-c100.log

-----------------------------------
Asynchronous Event Handling Latency
-----------------------------------
```

```
jvmsim disabled
Running 1000000 iterations
recording statistics...
handler thread exiting
Min: 2 us
Max: 10 us
Avg: 2.4395 us
StdDev: 0.5154 us
signal thread exiting

Criteria: latencies < 100
Result: PASS

Fri Oct 3 13:09:48 UTC 2008
The async_handler test appears to have completed.
```

Il test risulta essere stato eseguito con successo (Result: PASS), la latenza deve essere inferiore ai 100 us. Sul milione di cicli effettuati si è calcolata una latenza media di 2.4395 us, molto al di sotto del limite di 100 us

async_handler_jk

Simula un evento asincrono in una JVM real time. Si crea un thread server asincrono e lo si imposta a sleep in attesa che sia svegliato per effettuare qualche operazione. Si crea poi un thread utente che simula l'accadimento di un evento chiedendo al server di svolgere qualche operazione.

output `async_handler_jk-c100.log`

```
--- Running testcase async_handler_jk -c 100 ---
```

```
Fri Oct 3 13:09:48 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-async_handler_jk-c100.log
jvmsim disabled
delta = 12 us

Criteria: latencies < 100
Result: PASS

Fri Oct 3 13:09:48 UTC 2008
The async_handler_jk test appears to have completed.
```

Il test risulta essere stato eseguito con successo (Result: PASS)

pi_perf

Crea un thread ad alta priorità, uno a bassa priorità e diversi con priorità media. Il thread a priorità bassa possiede un PI lock, il thread ad alta priorità cerca di ottenere il lock. Il test misura la massima quantità di tempo che il thread ad alta priorità deve aspettare prima di ottenere il lock.

output pi_perf.log

```
--- Running testcase pi_perf -c 200 ---
Fri Oct 3 13:09:48 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-pi_perf- c200.log
Low prio thread started
High prio thread started
Busy 0 started
```

```
Busy 1 started
Time taken for high prio thread to get the lock once
released by low prio thread
Min wait time = 8 us
Max wait time = 11 us
Average wait time = 9.03 us
Standard Deviation = 0.30 us
Quantiles:
99.0% < 11
Low prio lock held time (min) = 10868 us
High prio lock wait time (max) = 10922 us
Criteria: High prio lock wait time < (Low prio lock held
time + 200 us)
Result: PASS

Fri Oct 3 13:09:50 UTC 2008
The pi_perf test appears to have completed.
```

Il test risulta essere stato eseguito con successo (Result: PASS)

pthread_kill_latency

Misura la latenza nell'inviare un segnale ad un thread utilizzando pthread_kill. Si creano due thread, uno (thread1) che riceve il segnale e l'altro (thread2) che lo invia. Prima di inviare il segnale il thread2 attende che thread1 sia inizializzato, annota il tempo e invia il segnale ptherad_kill a thread1. Sono riportati le latenze minima e massima.

Output pthread_kill_latency.log

```
--- Running testcase pthread_kill_latency -c 20 ---
```

```
Fri Oct 3 15:47:22 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-pthread_kill_latency-c20.log

--------------------------------
pthread_kill Latency
--------------------------------

Iterations: 10000
jvmsim disabled

Min: 3 us
Max: 36 us
Avg: 8.8997 us
StdDev: 3.4573 us
Quantiles:
99.0% < 20
99.9% < 26
99.99% < 36
Failures: 71
Criteria: Time < 20 us

Result: FAIL

Fri Oct 3 15:47:53 UTC 2008
The pthread_kill_latency test appears to have completed.
```

Questo test non risulta essere stato eseguito con successo. Il test definisce la variabile

THRESHOLD pari a 20 ns, ma solo nel 99% dei casi la latenza è inferiore ai 20 us. Nel 99.9% dei casi la latenza è inferiore ai 26 us e per il 99.99% dei casi la latenza è inferire a 36 us. Nel complesso 71 casi hanno superato la soglia di latenza fissata a 20 us.

prio-preempt

Testa la prelazione con priorità. Il thread principale crea diversi thread con priorità differenti, tutti i thread cercano di acquisire un mutex.

output prio-preempt

```
--- Running testcase prio-preempt -c 1 ---
Fri Oct 3 15:49:05 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-prio-preempt-c1.log
Maximum busy thread count(2), should not exceed number of
cpus(2)
Using 2

------------------
Priority Preemption
------------------

Busy Threads: 2
Interrupter Threads: Disabled
Worker Threads: 27

Busy Thread 2(81): Running...
Busy threads created!
Worker threads created
```

```
Signaling first thread
Busy Thread 2(81): Exiting

Criteria: All threads appropriately preempted within 1
loop(s)
Result: PASS

Fri Oct 3 15:49:06 UTC 2008
The prio-preempt test appears to have completed.
```

sched_latency

Misura il ritardo coinvolto in schedulazioni periodiche. Si crea un thread con priorità 89 e periodicamente viene interrotto per un certo periodo. Il ritardo, convertito in microsecondi, è misurato come:

delay = (now – start – i*period)

con:

- now = CLOCK_NONOTONIC gettime in ns
- start = CLOCK_NONOTONIC gettime all'inizio del test
- i = numero di iterazioni
- period = periodo scelto

Output sched_latency

```
--- Running testcase sched_latency -d 1 -t 5 -c 100 ---
Fri Oct 3 15:48:15 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-sched_latency-d1-t5-c100.log

------------------------------
```

```
Scheduling Latency
-------------------------------

Running 10000 iterations with a period of 5 ms
Periodic load duration: 1 ms
Expected running time: 50 s
jvmsim disabled

Start:  4 us: PASS
Min:   2 us: PASS
Max:   4 us: PASS
Avg:   3 us: PASS
StdDev: 0.1043 us
Quantiles:
99.0% < 3
99.9% < 4
99.99% < 4
Failed Iterations: 0

Criteria: latencies < 100 us
Result: PASS

Fri Oct 3 15:49:05 UTC 2008
The sched_latency test appears to have completed.
```

matrix_mult

Confronta i tempi impiegati ad effettuare la moltiplicazione sequenziale di matrici a quelli impiegati per moltiplicarle in parallelo in modo da poter stimare le performance del

processore. Il test è ripetuto per 100 volte e si calcola il tempo medio

output matrix_mult

```
--- Running testcase matrix_mult -c 0.75 ---
Fri Oct 3 15:47:53 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-matrix_mult-c0.75.log

----------------------------------------
Matrix Multiplication (SMP Performance)
----------------------------------------

Running 128 iterations
Matrix Dimensions: 100x100
Calculations per iteration: 8
Number of CPUs: 2
jvmsim disabled

Running sequential operations
Min: 72351 us
Max: 72387 us
Avg: 72381.0000 us
StdDev: 3.7582 us

Running concurrent operations (128x)
Min: 72122 us
Max: 72152 us
Avg: 36067.0000 us
StdDev: 5.1553 us
```

```
Concurrent Multipliers:
Min: 1.0032
Max: 1.0033
Avg: 2.0068

Criteria: 1.50 * average concurrent time < average
sequential time
Result: PASS

Fri Oct 3 15:48:07 UTC 2008
The matrix_mult test appears to have completed.
```

thread_clock (tc)

verifica se clock_gettime funziona correttamente. Crea un certo numero di thread che sono settati a sleep e un certo numero di thread che sono eseguiti. Legge i cicli di cpu dei thread e confronta la loro somma con i cicli di cpu del processo. Il test verifica se i cicli di cpu dei thread sleep è vicino a zero e la somma dei cicli di cpu di tutti i thread è comparabile con i cicli di cpu del processo.

Output thread_clock (tc)

```
--- Running testcase tc-2 -c 0.5 ---
Fri Oct 3 15:48:07 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-tc-2-c0.5.log
jvmsim disabled

5 sleeper threads created

2 worker threads created
```

```
Please wait...

Process: 14.3847 s
Threads: 14.3806 s
Delta:  0.0041 s

Criteria: Delta < 0.5000 s
Result: PASS

Fri Oct 3 15:48:15 UTC 2008
The tc-2 test appears to have completed.
```

gtod_latency

misura il tempo che intercorre tra varie chiamate a gettimeofday()

output gtod_latency

```
--- Running testcase gtod_latency ---
Fri Oct 3 15:46:39 UTC 2008
Logging      to      /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-gtod_latency.log

----------------------
Gettimeofday() Latency
----------------------
Iterations: 10000000

Min: 381 ns
Max: 5021 ns
```

```
Avg: 441.9066 ns
StdDev: 54.8412 ns
Quantiles:
99.0% < 401
99.9% < 546
99.99% < 1759
99.999% < 1854
99.9999% < 1944
99.99999% < 5021

Fri Oct 3 15:46:52 UTC 2008
The gtod_latency test appears to have completed.
```

sched_jitter

misura i jitter dovuti allo scheduling di processi real time

output

```
--- Running testcase sched_jitter ---
Fri Oct 3 15:49:06 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-sched_jitter.log

jvmsim disabled
delta: 527992565 ns
delta: 527967040 ns
delta: 527945859 ns
delta: 527814901 ns
...
delta: 527898904 ns
```

```
delta: 527819155 ns
delta: 527891879 ns
delta: 527958735 ns
delta: 527767571 ns
delta: 528034600 ns

max jitter: 733.583000 us

Fri Oct 3 15:57:55 UTC 2008
The sched_jitter test appears to have completed.
```

periodic_cpu_load

misura le variazioni nel tempo di esecuzione in vari periodi e priorità.

output

```
--- Running testcase periodic_cpu_load ---
Fri Oct 3 15:59:14 UTC 2008
Logging      to      /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-periodic_cpu_load.log

------------------------------------
Periodic CPU Load Execution Variance
------------------------------------

Running 6000 iterations per thread

Thread Group A:
 threads: 4
```

```
 priority: 63
 period: 40 ms

Thread Group B:
 threads: 4
 priority: 53
 period: 80 ms

Thread Group C:
 threads: 4
 priority: 43
 period: 160 ms

jvmsim disabled

TID 0 (A - prio 63) complete
TID 1 (A - prio 63) complete
TID 2 (A - prio 63) complete
TID 3 (A - prio 63) complete
TID 4 (B - prio 53) complete
TID 5 (B - prio 53) complete
TID 6 (B - prio 53) complete
TID 7 (B - prio 53) complete
TID 8 (C - prio 43) complete
TID 9 (C - prio 43) complete
TID 10 (C - prio 43) complete
TID 11 (C - prio 43) complete

Execution Time Statistics:
```

```
TID 0 (A)
 Min: 1290 us
 Max: 1294 us
 Avg: 1290.384033 us
 StdDev: 0.488738 us

 Quantiles:
99.0% < 1291
99.9% < 1291
Criteria: TID 0 did not miss a period
Result: PASS

TID 1 (A)
 Min: 1290 us
 Max: 1295 us
 Avg: 1290.320679 us
 StdDev: 0.470282 us

 Quantiles:
99.0% < 1291
99.9% < 1291
Criteria: TID 1 did not miss a period
Resul t: PASS

TID 2 (A)
 Min: 1292 us
 Max: 1295 us
 Avg: 1292.003174 us
```

```
 StdDev: 0.064471 us

 Quantiles:
99.0% < 1292
99.9% < 1293
Criteria: TID 2 did not miss a period
Result: PASS

TID 3 (A)
 Min: 1292 us
 Max: 1294 us
 Avg: 1292.002686 us
 StdDev: 0.054710 us

 Quantiles:
99.0% < 1292
99.9% < 1293
Criteria: TID 3 did not miss a period
Result: PASS

TID 4 (B)
 Min: 2389 us
 Max: 2392 us
 Avg: 2389.217529 us
 StdDev: 0.416563 us

 Quantiles:
99.0% < 2390
99.9% < 2391
```

```
Criteria: TID 4 did not miss a period
Result: PASS

TID 5 (B)
 Min: 2389 us
 Max: 2391 us
 Avg: 2389.236084 us
 StdDev: 0.427072 us

 Quantiles:
99.0% < 2390
99.9% < 2391
Criteria: TID 5 did not miss a period
Result: PASS

TID 6 (B)
 Min: 2389 us
 Max: 2391 us
 Avg: 2389.008789 us
 StdDev: 0.097063 us

 Quantiles:
99.0% < 2389
99.9% < 2390
Criteria: TID 6 did not miss a period
Result: PASS

TID 7 (B)
 Min: 2389 us
 Max: 2391 us
 Avg: 2389.248291 us
```

```
 StdDev: 0.436648 us

 Quantiles:
99.0% < 2390
99.9% < 2391
Criteria: TID 7 did not miss a period
Result: PASS

TID 8 (C)
 Min: 3446 us
 Max: 4539 us
 Avg: 3448.900635 us
 StdDev: 14.130017 us

 Quantiles:
99.0% < 3453
99.9% < 3454
Criteria: TID 8 did not miss a period
Result: PASS

TID 9 (C)
 Min: 3445 us
 Max: 3473 us
 Avg: 3448.631348 us
 StdDev: 3.525409 us

 Quantiles:
99.0% < 3458
99.9% < 3459
Criteria: TID 9 did not miss a period
```

```
Result: PASS

TID 10 (C)
 Min: 3445 us
 Max: 3495 us
 Avg: 3445.639648 us
 StdDev: 0.825019 us

 Quantiles:
99.0% < 3446
99.9% < 3450
Criteria: TID 10 did not miss a period
Result: PASS

TID 11 (C)
 Min: 3445 us
 Max: 3498 us
 Avg: 3445.999268 us
 StdDev: 0.714025 us

 Quantiles:
99.0% < 3446
99.9% < 3450
Criteria: TID 11 did not miss a period
Result: PASS

Fri Oct 3 16:15:14 UTC 2008
The periodic_cpu_load test appears to have completed.
```

periodic_cpu_load_single

misura le variazioni nel tempo di esecuzione in vari periodi, priorità e cicli.

output

```
--- Running testcase periodic_cpu_load_single ---
Fri Oct 3 16:15:15 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-periodic_cpu_load_single.log

------------------------------------
Periodic CPU Load Execution Variance
------------------------------------

Running 10000 iterations
priority: 90
 period: 5 ms
  loops: 1000
  logs: pcl*
jvmsim disabled

Execution Time Statistics:
Min: 2062 us
Max: 2067 us
Avg: 2062.0369 us
StdDev: 0.1855 us

Quantiles:
99.0% < 2063
99.9% < 2063
```

```
99.99% < 2067
Criteria: no missed periods
Result: PASS

Fri Oct 3 16:16:05 UTC 2008
The periodic_cpu_load_single test appears to have completed.
```

prio-wake

Testa il wakeup ordinato in base alla priorità

output

```
--- Running testcase prio-wake ---
Fri Oct 3 15:59:13 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-prio-wake.log
jvmsim disabled

-----------------------
Priority Ordered Wakeup
-----------------------
Worker Threads: 2

00000072 us: RealtimeThread-134541360 pri 004 started
00000112 us: RealtimeThread-134541656 pri 005 started
00000142 us: Master thread about to wake the workers
00000159 us: RealtimeThread-134541656 pri 005 awake
00000165 us: RealtimeThread-134541360 pri 004 awake
```

```
Criteria: Threads should be woken up in priority order
Result: PASS

Fri Oct 3 15:59:14 UTC 2008
The prio-wake test appears to have completed.
```

sched_football

Verifica che dei thread a priorità più bassa (la squadra che è in attacco) non riescano a prelazionare i thread a priorità più alta (la squadra che si difende)

output

```
--- Running testcase sched_football ---
Fri Oct 3 15:57:55 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-sched_football.log
jvmsim disabled
Running with: players_per_team=2 game_length=5
Starting 2 offense threads at priority 15
Starting 2 defense threads at priority 30
Starting referee thread
Game On (5 seconds)!
Game Over!
Final ball position: 0

Fri Oct 3 15:58:00 UTC 2008
```

testpi-0

verifica se l'ereditarietà delle priorità è presente

output testpi-0

```
--- Running testcase testpi-0 ---
Fri Oct 3 15:58:01 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-testpi-0.log
LIBC_VERSION: glibc 2.8
LIBPTHREAD_VERSION: NPTL 2.8
Prio inheritance support present

Fri Oct 3 15:58:01 UTC 2008
The testpi-0 test appears to have completed.
```

testpi-1

verifica l'ereditarietà delle priorità in due differenti situazioni. Nel primo caso verifica se la presenza dell'ereditarietà delle priorità permette ai thread con priorità più alta di essere eseguiti di più, quindi ripete il test senza l'ereditarietà della priorità

output testpi-1

```
--- Running testcase testpi-1 ---
Fri Oct 3 15:58:01 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-testpi-1.log
jvmsim disabled
Start ./testpi-1
protocol in mutexattr is 1
Thread 1003 started running with priority 0 on CPU
-1214115028

Thread 1003 at start pthread pol 0 pri 0 - Got global lock
```

```
Thread 1004 started running with prio 20 on CPU -1222507732
Thread 1005 started running with prio 30 on CPU -1230900436
Thread 1006 started running with prio 40 on CPU -1239293140
Noise Thread 1007 started running with prio 40 on CPU
-1247685844
Noise Thread 1007 loop 0 pthread pol 2 pri 40
Noise Thread 1007 loop 100 pthread pol 2 pri 40
Noise Thread 1007 loop 200 pthread pol 2 pri 40
...
Noise Thread 1007 loop 3100 pthread pol 2 pri 40
Noise Thread 1007 loop 3200 pthread pol 2 pri 40
Noise Thread 1007 loop 3300 pthread pol 2 pri 40
Thread 1003 loop 0 pthread pol 0 pri 0
Thread 1003 loop 100 pthread pol 0 pri 0
Thread 1003 loop 200 pthread pol 0 pri 0
Thread 1003 loop 300 pthread pol 0 pri 0
...
Thread 1003 loop 9800 pthread pol 0 pri 0
Thread 1003 loop 9900 pthread pol 0 pri 0
Noise Thread 1007 loop 3400 pthread pol 2 pri 40
Noise Thread 1007 loop 3500 pthread pol 2 pri 40
...
Noise Thread 1007 loop 4200 pthread pol 2 pri 40
Thread 1006 at start pthread pol 2 pri 40 - Got global lock
Thread 1006 loop 0 pthread pol 2 pri 40
...
Thread 1004 loop 800 pthread pol 2 pri 20
Thread 1004 loop 900 pthread pol 2 pri 20
```

```
Joining threads
Done
Criteria:Low Priority Thread should Preempt Higher Priority
Noise Thread

Fri Oct 3 15:58:09 UTC 2008
The testpi-1 test appears to have completed.
```

testpi-2

ripete il test precedente (testpi-1) introducendo un thread che genera rumore, disturbo e verifica se i thread a priorità più alta prelazionano i thread a priorità inferiore più volte.

Output testpi-2

```
--- Running testcase testpi-2 ---
Fri Oct 3 15:58:09 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-testpi-2.log
jvmsim disabled

Start ./testpi-2
protocol in mutexattr is 1
Thread 1026 started running with priority 10 on CPU
-1213852884
Thread 1026 at start pthread pol 2 pri 10 - Got global lock
Thread 1027 started running with prio 20 on CPU -1222245588
Thread 1028 started running with prio 30 on CPU -1230638292
Thread 1029 started running with prio 40 on CPU -1239030996
```

```
Noise Thread 1030 started running with prio 40 on CPU
-1247423700
Noise Thread 1030 loop 0 pthread pol 2 pri 40
Noise Thread 1030 loop 100 pthread pol 2 pri 40
Noise Thread 1030 loop 200 pthread pol 2 pri 40
...
Noise Thread 1030 loop 3200 pthread pol 2 pri 40
Noise Thread 1030 loop 3300 pthread pol 2 pri 40
Thread 1026 loop 0 pthread pol 2 pri 10
Thread 1026 loop 100 pthread pol 2 pri 10
...
Thread 1026 loop 2900 pthread pol 2 pri 10
Thread 1026 loop 3000 pthread pol 2 pri 10
Noise Thread 1030 loop 3400 pthread pol 2 pri 40
Noise Thread 1030 loop 3500 pthread pol 2 pri 40

Noise Thread 1030 loop 3600 pthread pol 2 pri 40
Thread 1026 loop 3100 pthread pol 2 pri 10
Thread 1026 loop 3200 pthread pol 2 pri 10
Thread 1026 loop 3300 pthread pol 2 pri 10
Noise Thread 1030 loop 3700 pthread pol 2 pri 40
Noise Thread 1030 loop 3800 pthread pol 2 pri 40
Noise Thread 1030 loop 3900 pthread pol 2 pri 40
Thread 1026 loop 3400 pthread pol 2 pri 10
Thread 1026 loop 3500 pthread pol 2 pri 10
Thread 1026 loop 3600 pthread pol 2 pri 10
...
Thread 1026 loop 9800 pthread pol 2 pri 10
```

```
Thread 1026 loop 9900 pthread pol 2 pri 10
Thread 1029 at start pthread pol 1 pri 40 - Got global lock
Thread 1029 loop 0 pthread pol 1 pri 40
Thread 1029 loop 100 pthread pol 1 pri 40
...
Thread 1028 loop 800 pthread pol 1 pri 30
Thread 1028 loop 900 pthread pol 1 pri 30
Thread 1027 at start pthread pol 2 pri 20 - Got global lock
Thread 1027 loop 0 pthread pol 2 pri 20
Thread 1027 loop 100 pthread pol 2 pri 20
...
Joining threads
Done

Criteria: Low Priority Thread and High Priority Thread
should prempt each other multiple times

Fri Oct 3 15:58:17 UTC 2008
The testpi-2 test appears to have completed.
```

testpi-4

ripete il test cambiando le politiche di schedulazione dei thread.

Output testpi-4

```
--- Running testcase testpi-4 ---
Fri Oct 3 15:58:17 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-testpi-4.log
```

```
jvmsim disabled

Start ./testpi-4
protocol in mutexattr is 1
Thread 1049 started running with priority 0 on CPU
-1214454996
Thread 1049 at start pthread pol 0 pri 0 - Got global lock
Thread 1050 started running with prio 20 on CPU -1222847700
Thread 1051 started running with prio 30 on CPU -1231240404
Thread 1052 started running with prio 40 on CPU -1239633108
Noise Thread started running with prio 40 on CPU -1248025812
Noise Thread 1053 loop 0 pthread pol 2 pri 40
Noise Thread 1053 loop 100 pthread pol 2 pri 40
...
Noise Thread 1053 loop 3300 pthread pol 2 pri 40
Thread 1049 loop 0 pthread pol 0 pri 0
Thread 1049 loop 100 pthread pol 0 pri 0
...
Noise Thread 1053 loop 4200 pthread pol 2 pri 40
Thread 1052 at start pthread pol 2 pri 40 - Got global lock
Thread 1052 loop 0 pthread pol 2 pri 40
...
Thread 1050 loop 800 pthread pol 2 pri 20
Thread 1050 loop 900 pthread pol 2 pri 20
Joining threads
Done

Fri Oct 3 15:58:25 UTC 2008
The testpi-4 test appears to have completed.
```

testpi-5

Il test utilizza l'ereditarietà delle priorità (PTREAD_PRIO_INHERIT), crea un thread figlio che tenta di acquisire un lock due volte.

otput testpi-5

```
--- Running testcase testpi-5 ---
Fri Oct 3 15:58:25 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-testpi-5.log
jvmsim disabled
Inside the timeout handler, killing the TC threads

Result:PASS

Fri Oct 3 15:58:45 UTC 2008
The testpi-5 test appears to have completed.
```

testpi-6

Usa il “robust mutex lock” (PTHREAD_MUTEX_ROBUST_NP) e il test-skeleton.

output testpi-6

```
--- Running testcase testpi-6 ---
Fri Oct 3 15:58:45 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-testpi-6.log
jvmsim disabled
Inside the timeout handler, killing the TC threads
```

```
Result:PASS


Fri Oct 3 15:59:05 UTC 2008
The testpi-6 test appears to have completed.
```

sbrk_mutex

utilizza NUM_THREADS per attraversare un vettore di mutex.

Output sbrk_mutex

```
--- Running testcase sbrk_mutex ---
Fri Oct 3 15:59:05 UTC 2008
Logging to /usr/src/ltp/testcases/realtime/logs/-
i686-2.6.23-2008-03-10-sbrk_mutex.log
jvmsim disabled
robust in mutexattr is 1
allocating and initializing 5000 mutexes
mutexes allocated and initialized successfully
joining threads

Fri Oct 3 15:59:13 UTC 2008
The sbrk_mutex test appears to have completed.
```

4.2.3 Test del sistema operativo

Dopo aver testato le caratteristiche real time del kernel, si è proceduto a testare il sistema operativo nella sua interezza. I test della suite che sono stati selezionati sono i seguenti:

- admin_tools
- timers
- sched
- mm

- math
- ipc
- dio
- fs

Ognuno di questi test è si appoggia su di un file avente lo stesso nome del test e contenente le istruzioni (i veri programmi di prova a cui saranno passati degli opportuni parametri) che saranno eseguiti. Per tutti i test si sono utilizzate le impostazioni di default, il comando eseguito è il seguente:

```
./runltp -f <test_name> -o <test_name>.log
```

dove <test_name> è il nome del file contenente i test da eseguire (es: ipc). Per ogni test eseguito sono di seguito riportati alcuni dei risultati ottenuti estratti dei file di log, in appendice sono invece riportati alcuni, tra i più brevi, dei file di comandi. Per ogni test eseguito sono stati generati in modo automatico oltre al file di log con l'output dell'elaborazione, un altro file di log contenente l'elenco dei test falliti.

admin_tools:

```
<<<test_start>>>
tag=su01 stime=1223321394
cmdline="export TCbin=$LTPROOT/testcases/bin;su01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
Machine type is: \n
sh: redhat: unknown operand
/usr/src/ltp/testcases/bin/su01: line 175:
     userdel: not    found
```

```
/usr/src/ltp/testcases/bin/su01: line 175:
      useradd: not    found
Could not add test user su_usr1.
<<<execution_status>>>
duration=1 termination_type=exited termination_id=1
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=cron_deny01 stime=1223321396
cmdline="cron_deny01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
sh: redhat: unknown operand
rm: cannot remove '/tmp/cron_deny_test': No such file or
          directory
rm: cannot remove '/tmp/cron_deny_test1': No such file or
          directory
mv: cannot rename '/var/spool/cron/deny': No such file or
          directory
/usr/src/ltp/testcases/bin/cron_deny01: line 182:
      userdel:   not found
/usr/src/ltp/testcases/bin/cron_deny01: line 182:
      userdel:   not found
/usr/src/ltp/testcases/bin/cron_deny01: line 182:
      useradd:   not found
```

```
Could not add test user cd_user1 to system.
<<<execution_status>>>
duration=1 termination_type=exited termination_id=1
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=cron_dirs_checks01 stime=1223321398
cmdline="cron_dirs_checks01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
Checking /etc/cron.daily
FAIL: /etc/cron.daily. Could not obtain directory status
/etc/cron.daily FAILED TEST!!!!!
Checking /etc/cron.hourly
FAIL: /etc/cron.hourly. Could not obtain directory status
/etc/cron.hourly FAILED TEST!!!!!
Checking /etc/cron.monthly
FAIL: /etc/cron.monthly. Could not obtain directory status
/etc/cron.monthly FAILED TEST!!!!!
Checking /etc/cron.weekly
FAIL: /etc/cron.weekly. Could not obtain directory status
/etc/cron.weekly FAILED TEST!!!!!
Checking /var/spool/cron
FAIL: /var/spool/cron. Could not obtain directory status
/var/spool/cron FAILED TEST!!!!!
```

```
<<<execution_status>>>
duration=0 termination_type=exited termination_id=1
     corefile=no
cutime=0 cstime=0
<<<test_end>>>
```

I test contenuti nel file admin_tools non sono stati eseguiti tutti in modo corretto. Alcuni test hanno fallito. Dall'analisi del log è stato possibile risalire ai motivi che hanno portato al fallimento del test. Alcuni test hanno bisogno della presenza nel sistema degli eseguibili userdel e useradd, mentre busybox mette a disposizione dell'amministratore i programmi deluser e adduser. Altra causa di errore è dovuta al fatto che il sistema testato è stato progettato per essere eseguito come sistema embedded, quindi non ha la varietà di file che è possibile trovare su di un sistema desktop o server

dio:

```
<<<test_start>>>
tag=dio01 stime=1223323740
cmdline="diotest1"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
diotest01  1 CONF : O_DIRECT is not supported by this
           filesystem.
<<<execution_status>>>
duration=1 termination_type=exited termination_id=0
           corefile=no
cutime=0 cstime=0
```

```
<<<test_end>>>

<<<test_start>>>
tag=dio02 stime=1223323741
cmdline="diotest2"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
diotest02  1 CONF : O_DIRECT is not supported by this
           filesystem.
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
           corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=dio03 stime=1223323741
cmdline="diotest3"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
diotest03  1 CONF : O_DIRECT is not supported by this
         filesystem.
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
```

```
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=dio04 stime=1223323741
cmdline="diotest4"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
diotest4  1 CONF : O_DIRECT is not supported by this
           filesystem.
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
           corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=dio05 stime=1223323741
cmdline="diotest5"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
diotest05  1 CONF : O_DIRECT is not supported by this
           filesystem.
```

```
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
           corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=dio06 stime=1223323741
cmdline="diotest6"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
diotest06  1 CONF : O_DIRECT is not supported by this
           filesystem.
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
      corefile=no
cutime=0 cstime=0
<<<test_end>>>
diotest01  1 CONF : O_DIRECT is not supported by this
           filesystem.

<<<test_start>>>
tag=dio07 stime=1223323741
cmdline="diotest1 -b 65536"
contacts=""
analysis=exit
```

```
initiation_status="ok"
<<<test_output>>>
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=dio29 stime=1223323792
cmdline="diotest3 -b 65536 -n 100 -i 1000 -o 1024000"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
           corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=dio30 stime=1223323792
cmdline="diotest6 -b 65536 -n 100 -i 1000 -o 1024000"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
```

```
diotest06  1 CONF : O_DIRECT is not supported by this
           filesystem.
incrementing stop
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
           corefile=no
cutime=0 cstime=0
<<<test_end>>>
```

Il file di log contenente i test falliti è risultato vuoto.

fs:

```
<<<test_start>>>
tag=iogen01 stime=1223322850
cmdline="export LTPROOT; rwtest -N iogen01 -i 120s -s  \
     read,write -Da -Dv -n 2                          \
     500b:doio.f1.$$ 1000b:doio.f2.$$"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
/usr/src/ltp/testcases/bin/iogen -N iogen01 -i 120s -s  \
     read,write 500b:doio.f1.30692 1000b:doio.f2.30692 | \
     /usr/src/ltp/testcases/bin/doio -N iogen01 -a -v  \
     -n 2 -k

iogen(iogen01) starting up with the following:

Out-pipe:       stdout
```

```
Iterations:      120 seconds
Seed:         30699
Offset-Mode:      sequential
Overlap Flag:     off
Mintrans:       1      (1 blocks)
Maxtrans:       131072   (256 blocks)
O_RAW/O_SSD Multiple: (Determined by device)
Syscalls:       read write
Aio completion types: none
Flags:         buffered sync

Test Files:

Path              Length iou      raw iou file
               (bytes)     (bytes) (bytes) type
-------------------------------------------------------------
/tmp/ltp-30588/doio.f1.30692  256000        1   512 regular
/tmp/ltp-30588/doio.f2.30692  512000  1   512 regular
iogen01   1 PASS : Test passed
Test passed
<<<execution_status>>>
duration=121 termination_type=exited termination_id=0
     corefile=no
cutime=4506 cstime=18023
<<<test_end>>>

<<<test_start>>>
tag=fs_inod01 stime=1223322971
cmdline="fs_inod $TMP 10 10 1"
```

```
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
FS_INODE: File system stress - inode allocation/deallocation
Volume under test: /tmp/ltp-30588
Number of subdirectories: 10
Number of files: 10
Number of loops: 1
Execution begins
     Mon Oct 6 19:56:11 UTC 2008
================================================
MULTIPLE PROCESSES CREATING AND DELETING FILES
================================================
/usr/src/ltp/testcases/bin/fs_inod: creating dir2
     subdirectories
/usr/src/ltp/testcases/bin/fs_inod: mkdir dir0
...
/usr/src/ltp/testcases/bin/fs_inod: mkdir dir9
/usr/src/ltp/testcases/bin/fs_inod: creating dir1
     subdirectories & files
...
/usr/src/ltp/testcases/bin/fs_inod: mkdir dir9
/usr/src/ltp/testcases/bin/fs_inod: touch files
     [0-10]/file10[0-10]
Executing loop 1 of 1...
/usr/src/ltp/testcases/bin/fs_inod: cd ../dir1 & creating
     files
```

```
/usr/src/ltp/testcases/bin/fs_inod: touch files
     [0-10]/file10[0-10]
...
Execution completed
Mon Oct 6 19:56:11 UTC 2008
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=3 cstime=5
<<<test_end>>>

<<<test_start>>>
tag=stream05 stime=1223322972
cmdline="stream05"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
stream05  1 PASS : Test passed in block0.
stream05  2 PASS : Test passed in block1.
stream05  3 PASS : Test passed in block2.
stream05  4 PASS : Test passed in block3.
stream05  5 PASS : Test passed in block4.
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>
```

```
<<<test_start>>>
tag=ftest07 stime=1223322976
cmdline="ftest07"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
ftest07   1 PASS : Test passed.
<<<execution_status>>>
duration=4 termination_type=exited termination_id=0
     corefile=no
cutime=5 cstime=470
<<<test_end>>>

<<<test_start>>>
tag=ftest08 stime=1223322980
cmdline="ftest08"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
ftest08   1 PASS : Test passed.
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=3 cstime=52
<<<test_end>>>

<<<test_start>>>
```

```
tag=lftest01 stime=1223322980
cmdline="lftest 100"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
Started building a 100 megabyte file @ Mon Oct
     6 19:56:20      2008

.....................................................
Finished building a 100 megabyte file @ Mon Oct
     6 19:56:20      2008

Number of Writes: 100
Number of Seeks: 100
Total time for test to run: 0 minute(s) and 0 seconds
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=7
<<<test_end>>>
```

Anche per i test eseguiti sul file system non sono stati riportati fallimenti

ipc:

```
<<<test_start>>>
tag=pipeio_3 stime=1223321218
cmdline="pipeio -T pipeio_3 -c 5 -s 4090 -i 100 -u \
```

```
     -b -f x80"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
pipeio_3  1 PASS : 1 PASS 486 pipe reads complete, read
     size = 4090, sys pipe,
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=1 cstime=1
<<<test_end>>>

<<<test_start>>>
tag=pipeio_4 stime=1223321218
cmdline="pipeio -T pipeio_4 -c 5 -s 4090 -i 100 -u -f x80"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
pipeio_4  1 PASS : 1 PASS 486 pipe reads complete,
     read  size = 4090, sys pipe,
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=1 cstime=2
<<<test_end>>>

<<<test_start>>>
```

```
tag=shmem_test_07 stime=1223321239
cmdline="       shmem_test_07"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
shmem_test_07: IPC Shared Memory TestSuite program
     Number of writers  = 2
     Number of readers  = 2
     Bytes per writer = 200000
          writer (000): shared memory checksum 01850160
          reader (000) of writer (000): checksum 01850160
          reader (001) of writer (000): checksum 01850160
          writer (001): shared memory checksum 018501a0
          reader (000) of writer (001): checksum 018501a0
          reader (001) of writer (001): checksum 018501a0

     Main: readers calculated segment successfully

successful!
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=1 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=signal_test_03 stime=1223321243
cmdline="       signal_test_03"
```

```
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
signal_test_03: IPC Signals TestSuite program

     (BEGIN) Critial section

     (END) Critial section
     received signal: (SIGILL)

successful!
<<<execution_status>>>
duration=1 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=signal_test_06 stime=1223321248
cmdline="       signal_test_06"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
signal_test_06: IPC TestSuite program
     Block all signals from interrupting the process
     Send MAX (1048576) SIGUSR1 signals to the process...
```

```
      Ensure at least one SIGUSR1 signal is pending

      Change signal mask & wait for SIGUSR1 signal
      caught SIGUSR1 (10) signal

successful!
<<<execution_status>>>
duration=2 termination_type=exited termination_id=0
      corefile=no
cutime=5 cstime=26
<<<test_end>>>

<<<test_start>>>
tag=signal_test_07 stime=1223321250
cmdline="       signal_test_07"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
signal_test_07: IPC TestSuite program

      Send MAX (1048576) signals to the process...
      Received EVERY signal!

successful!
incrementing stop
<<<execution_status>>>
duration=2 termination_type=exited termination_id=0
      corefile=no
```

```
cutime=36 cstime=113
<<<test_end>>>
```

nel file di log non sono riportati errori nell'esecuzione dei test

math:

```
<<<test_start>>>
tag=abs01 stime=1223323830
cmdline="abs01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
abs01    1 PASS : Test passed
abs01    2 PASS : Test passed
abs01    3 PASS : Test passed
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=atof01 stime=1223323830
cmdline="atof01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
```

```
atof01   1 PASS : Test passed
atof01   2 PASS : Test passed
atof01   3 PASS : Test passed
atof01   4 PASS : Test passed
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=float_bessel stime=1223323830
cmdline="cd $LTPROOT/testcases/bin; float_bessel -v"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
float_bessel 0 INFO : float_bessel: will run for 500 loops
float_bessel 0 INFO : float_bessel: using . as data directory
float_bessel 0 INFO : float_bessel: will run 5 functions, 20
     threads per function
float_bessel 0 INFO : signal handler 3085036432 started
float_bessel 0 INFO : Signal handler starts waiting...
float_bessel 0 INFO  : Initial thread: Waiting for 100
      threads to finish
float_bessel 0 INFO : thread 0 (j0) terminated successfully
      500 loops.
...
```

```
float_bessel 0 INFO : thread 19 (j0) terminated
      successfully 500 loops.
float_bessel  0 INFO : thread 20 (j1) terminated
      successfully 500 loops.
...
float_bessel  0 INFO : thread 39 (j1) terminated
      successfully 500 loops.
float_bessel  0 INFO : thread 40 (y0) terminated
      successfully 500 loops.
...
float_bessel  0 INFO : thread 59 (y0) terminated
      successfully 500 loops.
float_bessel  0 INFO : thread 60 (y1) terminated
      successfully 500 loops.
float_bessel  0 INFO : thread 61 (y1) terminated
      successfully 500 loops.
...
float_bessel  0 INFO : thread 79 (y1) terminated
      successfully 500 loops.
float_bessel  0 INFO : thread 80 (lgamma) terminated
      successfully 500 loops.
...
float_bessel  0 INFO : thread 98 (lgamma) terminated
      successfully 500 loops.
float_bessel  0 INFO : thread 99 (lgamma) terminated
      successfully 500 loops.
float_bessel  1 PASS : Test passed
<<<execution_status>>>
```

```
duration=7 termination_type=exited termination_id=0
      corefile=no
cutime=1348 cstime=27
<<<test_end>>>
```

come per gli altri test, anche math è stato eseguito in modo corretto

mm:

```
<<<test_start>>>
tag=mm01 stime=1223322545
cmdline="mmap001 -m 10000"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
mmap001   0 INFO : mmap()ing file of 10000 pages or
     40960000 bytes
mmap001   1 PASS : mmap() completed successfully.
mmap001   0 INFO : touching mmaped memory
mmap001   2 PASS : we're still here, mmaped area must
     be good
mmap001   3 PASS : msync() was successful
mmap001   4 PASS : munmap() was successful
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=14 cstime=5
<<<test_end>>>
```

```
<<<test_start>>>
tag=mtest01w stime=1223322545
cmdline="mtest01 -p80 -w"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
mtest01   0 INFO : Total memory used needed to reach
     maxpercent = 2465430 kbytes
mtest01   0 INFO : Total memory already used on system
     = 31640 kbytes
mtest01   0 INFO : Filling up 800f ram which is 2433790
     kbytes
mtest01   1 PASS : 2433790 kbytes allocated and used.
<<<execution_status>>>
duration=5 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=mtest01w stime=1223322545
cmdline="mtest01 -p80 -w"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
mtest01   0 INFO : Total memory used needed to reach
```

```
      maxpercent = 2465430 kbytes
mtest01   0 INFO : Total memory already used on system
      = 31640 kbytes
mtest01   0 INFO : Filling up 800f ram which is 2433790
      kbytes
mtest01   1 PASS : 2433790 kbytes allocated and used.
<<<execution_status>>>
duration=5 termination_type=exited termination_id=0
      corefile=no
cutime=0 cstime=0
<<<test_end>>>
```

Il test non ha generato errori

sched:

```
<<<test_start>>>
tag=pth_str01 stime=1223322429
cmdline="pth_str01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
pth_str01  0 INFO : Allocating 85 nodes.
pth_str01  0 INFO : Creating root thread attributes via
       pthread_attr_init.
pth_str01  0 INFO : Creating root thread via
       pthread_create.
pth_str01  0 INFO : thread 0 creating kids, cdepth=1
```

```
pth_str01  0 INFO : thread 1 started
pth_str01  0 INFO : thread 2 started
pth_str01  0 INFO : thread 3 started
pth_str01  0 INFO : thread 4 started
pth_str01  0 INFO : thread 4 creating kids, cdepth=2
pth_str01  0 INFO : thread 1 creating kids, cdepth=2
...
pth_str01  0 INFO : thread 20 started
pth_str01  0 INFO : thread 20 creating kids, cdepth=3
pth_str01  0 INFO : thread 5 creating kids, cdepth=3
...
pth_str01  0 INFO : thread 84 started
pth_str01  0 INFO : thread 84 is a leaf node, depth=4
...
pth_str01  0 INFO : thread 50 is a leaf node, depth=4
pth_str01  0 INFO : thread 49 exiting, depth=4,
      status=0, addr=0x804f030
...
pth_str01  0 INFO : thread 0 exiting, depth=4,
      status=0, addr=0x804d008
pth_str01  1 PASS : Test passed
<<<execution_status>>>
duration=1 termination_type=exited termination_id=0
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=trace_sched01 stime=1223322430
```

```
cmdline="  trace_sched -c 1"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
For processor number = 0
============================
pid of task = 6761 priority requested = 14 priority assigned
      by scheduler = 29
pid of task = 6761 priority requested = 57 priority assigned
      by scheduler = 57
pid of task = 6761 priority requested = 57 priority assigned
      by scheduler = 57
...
pid of task = 6761 priority requested = 50 priority assigned
      by scheduler = 50
pid of task = 6761 priority requested = 50 priority assigned
      by scheduler = 50
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=hackbench01 stime=1223322430
cmdline="hackbench 150 process 1000"
contacts=""
```

```
analysis=exit
initiation_status="ok"
<<<test_output>>>
Running with 150*40 (== 6000) tasks.
SENDER: write (error: Connection reset by peer)
...
SENDER: write (error: Connection reset by peer)
SENDER: write (error: Broken pipe)
...
<<<execution_status>>>
duration=8 termination_type=exited termination_id=1
      corefile=no
cutime=1 cstime=29
<<<test_end>>>
```

Nessun errore è riportato nel file di log

syscalls:

```
<<<test_start>>>
tag=accept01 stime=1223321857
cmdline="accept01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
accept01  1 PASS : bad file descriptor successful
accept01  2 PASS : bad file descriptor successful
accept01  3 PASS : invalid socket buffer successful
```

```
accept01  4 PASS : invalid salen successful
accept01  5 PASS : invalid salen successful
accept01  6 PASS : no queued connections successful
accept01  7 PASS : UDP accept successful
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=access01 stime=1223321857
cmdline="access01"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
access01  1 PASS : access(accessfile, F_OK)      returned 0
access01  2 PASS : access(accessfile, X_OK) returned 0
access01  3 PASS : access(accessfile, W_OK) returned 0
access01  4 PASS : access(accessfile, R_OK) returned 0
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=access02 stime=1223321857
```

```
cmdline="access02"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
access02  1 PASS : Functionality of access() for
     test_file1, successful
access02  2 PASS : Functionality of access() for
     test_file2, successful
access02  3 PASS : Functionality of access() for
     test_file3, successful
access02  4 PASS : Functionality of access() for
     sym_file, successful
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=alarm05 stime=1223321857
cmdline="alarm05"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
alarm05   1 PASS : Functionality of alarm(5) successful
<<<execution_status>>>
duration=8 termination_type=exited termination_id=0
```

```
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=chdir03 stime=1223321886
cmdline="chdir03"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
chdir03   1 PASS : expected failure - errno = 13 :
      Permission denied
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
      corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=creat07 stime=1223321887
cmdline="creat07 -F $LTPROOT/testcases/bin/test1"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
<program name unknown>: error while loading shared
      libraries: libc.so.6: cannot open shared object file:
       No   such file or directory
```

```
creat07   1 FAIL : Failures reported above
<<<execution_status>>>
duration=0 termination_type=exited termination_id=1
     corefile=no
cutime=1 cstime=0
<<<test_end>>>
```

L'esecuzione del syscalls, ha generato il fallimento di alcuni test. Dall'analisi del log sembra che alcuni programmi non riescano a trovare la libreria condivisa libc. Sono necessarie ulteriori indagini per verificare se è un problema ascrivibile alla configurazione della test suite o del sistema. Ad una prima analisi si tenderebbe ad escludere un difetto nella distribuzione vista l'importanza della libreria c, praticamente utilizzata da tutti gli altri eseguibili presenti nel sistema.

timers:

```
<<<test_start>>>
tag=clock_gettime02 stime=1223322417
cmdline="clock_gettime02"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
clock_gettime02  1 PASS : clock_gettime(2) Passed
clock_gettime02  2 PASS : clock_gettime(2) Passed
<<<execution_status>>>
duration=0    termination_type=exited    termination_id=0
corefile=no
```

```
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=clock_gettime03 stime=1223322417
cmdline="clock_gettime03"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
clock_gettime03  1 PASS : clock_gettime(2) expected
     failure; Got errno - EFAULT : Bad address
clock_gettime03  2 PASS : clock_gettime(2) expected
     failure; Got errno - EFAULT : Bad address
clock_gettime03  3 PASS : clock_gettime(2) expected
     failure; Got errno - EINVAL : Invalid parameter
clock_gettime03  4 PASS : clock_gettime(2) expected
     failure; Got errno - EINVAL : Invalid parameter
clock_gettime03  5 PASS : clock_gettime(2) expected
     failure; Got errno - EFAULT : Bad address
clock_gettime03  6 PASS : clock_gettime(2) expected
     failure; Got errno - EFAULT : Bad address
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>
```

```
<<<test_start>>>
tag=clock_settime02 stime=1223322417
cmdline="clock_settime02"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
clock_settime02  1 PASS : clock_settime(2) Passed
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=clock_settime03 stime=1223322417
cmdline="clock_settime03"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
clock_settime03  1 PASS : clock_settime(2) expected
     failure; Got errno - EFAULT : Bad address
clock_settime03  2 PASS : clock_settime(2) expected
     failure; Got errno - EINVAL : Invalid parameter
clock_settime03  3 PASS : clock_settime(2) expected
     failure; Got errno - EINVAL : Invalid parameter
clock_settime03  4 PASS : clock_settime(2) expected
```

```
     failure; Got errno - EINVAL : Invalid parameter
clock_settime03  5 PASS : clock_settime(2) expected
     failure; Got errno - EINVAL : Invalid parameter
clock_settime03  6 PASS : clock_settime(2) expected
     failure; Got errno - EINVAL : Invalid parameter
clock_settime03  7 PASS : clock_settime(2) expected
     failure; Got errno - EPERM : Operation not permitted
clock_settime03  8 PASS : clock_settime(2) expected
     failure; Got errno - EFAULT : Bad address
clock_settime03  9 PASS : clock_settime(2) expected
     failure; Got errno - EFAULT : Bad address
<<<execution_status>>>
duration=1 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>

<<<test_start>>>
tag=timer_create02 stime=1223322418
cmdline="timer_create02"
contacts=""
analysis=exit
initiation_status="ok"
<<<test_output>>>
timer_create02  1 PASS : CLOCK_REALTIME with
     notification type SIGEV_SIGNAL
timer_create02  2 PASS : CLOCK_MONOTONIC with
     notification type SIGEV_SIGNAL
```

```
timer_create02  3 PASS : CLOCK_PROCESS_CPUTIME_ID with
     notification type SIGEV_SIGNAL
timer_create02  4 PASS : CLOCK_THREAD_CPUTIME_ID with
     notification type SIGEV_SIGNAL
timer_create02  5 PASS : CLOCK_REALTIME with
     notification type NULL
timer_create02  6 PASS : CLOCK_MONOTONIC with
     notification type NULL
timer_create02  7 PASS : CLOCK_PROCESS_CPUTIME_ID with
     notification type NULL
timer_create02  8 PASS : CLOCK_THREAD_CPUTIME_ID with
     notification type NULL
timer_create02  9 PASS : CLOCK_REALTIME with
     notification type SIGEV_NONE
timer_create02  10 PASS : CLOCK_MONOTONIC with
     notification type SIGEV_NONE
timer_create02  11 PASS : CLOCK_PROCESS_CPUTIME_ID with
     notification type SIGEV_NONE
timer_create02  12 PASS : CLOCK_THREAD_CPUTIME_ID with
     notification type SIGEV_NONE
<<<execution_status>>>
duration=0 termination_type=exited termination_id=0
     corefile=no
cutime=0 cstime=0
<<<test_end>>>
```

Capitolo 5
Conclusioni

Il mercato dei dispositivi embedded è enorme. Dalla semplice lavapiatti ad un sofisticato satellite per le telecomunicazioni, ogni attività umana che ha una qualche relazione con un apparato elettronico fa certamente uso di un dispositivo embedded. Attualmente utilizzare Linux per la realizzazione di un sistema embedded può essere considerata la migliore scelta possibile. Ormai da qualche anno Linux è il kernel di riferimento per la creazione di sistemi embedded. È riuscito a sostituire i sistemi operativi proprietari che fino a poco tempo fa dominavano questo mercato e ciò grazie ad una serie di vantaggi che, insieme al software libero, offre. Questi vantaggi sono sia di natura tecnica che economica, alcuni sono evidenti, altri coinvolgono aspetti più filosofici o personali, ma tutti hanno come conseguenza la scelta di Linux e del software libero. Da un punto di vista economico, Linux grazie alla licenza GPL, permette di eliminare i costi relativi alle royalty da corrispondere a terze parti. Da un punto di vista strettamente tecnico permette la libera circolazione e modifica del codice secondo le proprie necessità. Linux offre modularità, stabilità e robustezza, permette quindi la costruzione di sistemi estremamente affidabili e configurabili su misura. I dispositivi embedded offrono hardware sempre più complessi che devono essere gestiti in modo ottimale. Spesso sono dotati di grandi quantità di memoria, molte volte con display a colori o con touch screen e offrono all'utente una gran quantità di applicativi. Ciò costituisce un ulteriore vantaggio di Linux rispetto ai classici sistemi operativi embedded. I sistemi operativi tradizionalmente utilizzati per i sistemi embedded non dispongono di tecniche avanzate per la gestione della memoria, invece Linux può offrire la protezione della memoria, permettendo non solo la separazione dello spazio del kernel dallo spazio utente, ma anche la separazione della memoria occupata dai singoli processi. La gran disponibilità di driver per quasi tutto l'hardware esistente evita di dover reinventare la ruota per ogni nuovo dispositivo prodotto. Un grande impulso allo

sviluppo e all'adozione di linux per i sistemi embedded è stato dato da società come MontaVista con la loro soluzione real time del kernel. L'importanza delle patch di MontaVista al kernel non risiedono soltanto nelle innovazioni tecniche, ma anche nel fatto che hanno preservato il modello di programmazione di Linux, a differenza delle soluzioni adottate da RTLinux e RTAI che prevedono l'introduzione di nuove API per la creazione e la gestione dei processi e dei task real time. I vantaggi di un kernel real time facilmente accessibile tramite le classiche chiamate di sistema hanno immediate ricadute, oltre che per i processi industriali, anche sui sistemi desktop. I sistemi desktop ormai richiedono sia il multi processing che il multi programming. Gli utenti vogliono poter usufruire di contenuti multimediali di qualità sempre maggiore, scaricandoli direttamente dalla rete o accedendovi dalle proprie macchine. Spesso queste stesse macchine devono garantire, contemporaneamente alla riproduzione dell'evento multimediale, la gestione di account di posta elettronica o messaggistica istantanea o programmi di video conferenze. Da ciò si intuisce l'importanza dell'innovazione introdotta da MontaVista. [54]. Un altro dei motivi che hanno favorito l'adozione di Linux per lo sviluppo dei sistemi embedded è legato alla sua grande diffusione e alla vasta schiera di tecnici, informatici e ingegneri che lo supportano. Per le industrie il dover supportare molte piattaforme proprietarie si traduce in un aumento dei costi, bisogna dedicare risorse tecniche e umane per amministrare e gestire sistemi operativi e applicativi tra di loro incompatibili. Linux garantisce invece una piattaforma unica e altamente scalabile. Quelle elencate sono delle motivazioni tecniche ed economiche che possono spiegare l'uso di Linux da parte di aziende e società, ma non rispondono al perché in pochi anni Linux e il software libero siano riusciti a produrre codice di elevata qualità e ad affermarsi tra gli addetti ai lavori. Perché linux piace agli ingegneri e agli informatici? A questa domanda ha provato a rispondere Eric Raymond nei suoi saggi, in particolare in “Colonizzare la noosfera” [56]. Raymond conclude che l'impulso che spinge gli hacker a lavorare su progetti open source o free software, a produrre codice di alta qualità per poi farne dono all'intera comunità non ha alla base nessuna motivazione di carattere economico o materiale. Gli hacker ricavano

soddisfazione personale per aver realizzato un software complesso che funziona. Inoltre contribuiscono, coscientemente o meno, allo sviluppo della comunità con il solo intento di acquisire prestigio. Hanno lo scopo finale di essere riconosciuti come pari fra pari, e il solo modo che hanno a disposizione per acquisire prestigio è quello di produrre e rilasciare software che possa essere giudicato e apprezzato prima di tutto dai membri della comunità e in secondo luogo da tutti i potenziali utilizzatori del prodotto. Questa filosofia ha portato alla produzione di software di qualità elevatissima, alla diffusione e alla condivisione di conoscenza e come "effetti collaterali" ha permesso la nascita di nuove realtà economiche (Red Hat, MontaVista, Google) e il rilancio di storiche compagnie (IBM, Sun, Novel). Queste società hanno intuito la grande potenzialità del software a sorgente aperto investendo in esso e rilasciando a loro volta alcuni dei loro prodotti di punta, (Java, Solaris, OpenOffice.org, Netscape/Mozilla/Firefox, Eclipse) alla comunità del software libero. In virtù di questi fatti, la decisione di utilizzare il sistema operativo Gnu/Linux per sviluppare i propri progetti appare una ottima scelta.

Appendice A – Dipendenze del software

Per rilevare da quali librerie condivise (*.so) i vari software installati dipendessero è stato utilizzato il programma *ldd*, che chiamato sugli eseguibili generati dalle varie compilazioni, ha prodotto i seguenti risultati:

NtpClient

```
# ldd ntpclient
        linux-gate.so.1 => (0x0012c000)
        librt.so.1 => /lib/librt.so.1 (0x0012d000)
        libc.so.6 => /lib/libc.so.6 (0x00136000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x0026d000)
        /lib/ld-linux.so.2 (0x00110000)
```

Bash

```
# ldd bash
        linux-gate.so.1 => (0x00110000)
        libncurses.so.5 => /lib/libncurses.so.5 (0x051a6000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libtinfo.so.5 => /lib/libtinfo.so.5 (0x05172000)
        /lib/ld-linux.so.2 (0x006da000)
```

Gdb

```
# ldd gdbserver
        linux-gate.so.1 => (0x00110000)
        libthread_db.so.1 => /lib/libthread_db.so.1 (0x00111000)
```

```
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        /lib/ld-linux.so.2 (0x006da000)
```

BusyBox

```
# ldd busybox
        linux-gate.so.1 => (0x00110000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libm.so.6 => /lib/libm.so.6 (0x00869000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        /lib/ld-linux.so.2 (0x006da000)
```

OpenSSh

```
# ldd scp
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)

# ldd sftp
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
```

```
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)

# ldd ssh
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)

# ldd ssh-add
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
```

```
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)

# ldd ssh-agent
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)

# ldd ssh-keygen
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)
```

```
# ldd ssh-keyscan
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)

# ldd sshd
        linux-gate.so.1 => (0x00110000)
        libresolv.so.2 => /lib/libresolv.so.2 (0x00de2000)
        libcrypto.so.7 => /lib/libcrypto.so.7 (0x0402a000)
        libutil.so.1 => /lib/libutil.so.1 (0x0052a000)
        libz.so.1 => /lib/libz.so.1 (0x008b6000)
        libnsl.so.1 => /lib/libnsl.so.1 (0x004e9000)
        libcrypt.so.1 => /lib/libcrypt.so.1 (0x04f17000)
        libc.so.6 => /lib/libc.so.6 (0x006fe000)
        libdl.so.2 => /lib/libdl.so.2 (0x00894000)
        /lib/ld-linux.so.2 (0x006da000)
```

Kbd

```
# ldd dumpkeys getkeycodes kbd_mode kbdrate          \
        loadkeys loadunimap mapscrn openvt resizecons    \
        setfont setkeycodes setleds setmetamode          \
```

```
        showconsolefont showkey
dumpkeys:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
getkeycodes:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
kbd_mode:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
kbdrate:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
loadkeys:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
loadunimap:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
mapscrn:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
```

```
        /lib/ld-linux.so.2 (0x00980000)
openvt:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
resizecons:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
setfont:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
setkeycodes:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
setleds:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
setmetamode:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
showconsolefont:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
```

```
        /lib/ld-linux.so.2 (0x00980000)
showkey:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
```

Mtd utils

```
# ldd docfdisk doc_loadbios flashcp flash_erase        \
   flash_eraseall flash_info flash_lock
docfdisk:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
doc_loadbios:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
flashcp:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
flash_erase:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
flash_eraseall:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
```

```
        /lib/ld-linux.so.2 (0x00980000)
flash_info:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
flash_lock:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)

# ldd flash_otp_dump flash_otp_info flash_unlock       \
   ftl_check ftl_format jffs2dump
flash_otp_dump:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
flash_otp_info:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
flash_unlock:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
ftl_check:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
```

```
ftl_format:
      linux-gate.so.1 => (0x00110000)
      libc.so.6 => /lib/libc.so.6 (0x009a0000)
      /lib/ld-linux.so.2 (0x00980000)
jffs2dump:
      linux-gate.so.1 => (0x00110000)
      libc.so.6 => /lib/libc.so.6 (0x009a0000)
      /lib/ld-linux.so.2 (0x00980000)

# ldd mkfs.jffs mkfs.jffs2 mtd_debug nanddump          \
      nandwrite nftldump nftl_format rfdformat         \
      rfdformat sumtool
mkfs.jffs:
      linux-gate.so.1 => (0x00110000)
      libc.so.6 => /lib/libc.so.6 (0x009a0000)
      /lib/ld-linux.so.2 (0x00980000)
mkfs.jffs2:
      linux-gate.so.1 => (0x00110000)
      libz.so.1 => /lib/libz.so.1 (0x00b58000)
      liblzo2.so.2 => /usr/lib/liblzo2.so.2 (0x00422000)
      libc.so.6 => /lib/libc.so.6 (0x009a0000)
      /lib/ld-linux.so.2 (0x00980000)
mtd_debug:
      linux-gate.so.1 => (0x00110000)
      libc.so.6 => /lib/libc.so.6 (0x009a0000)
      /lib/ld-linux.so.2 (0x00980000)
nanddump:
      linux-gate.so.1 => (0x00110000)
```

```
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
nandwrite:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
nftldump:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
nftl_format:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
rfdformat:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
rfdformat:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
sumtool:
        linux-gate.so.1 => (0x00110000)
        libc.so.6 => /lib/libc.so.6 (0x009a0000)
        /lib/ld-linux.so.2 (0x00980000)
```

i tool contenuti nel pacchetto mtd-utils sono tutti linkati alle librerie C, ad eccezione

dell'eseguibile mkfs.jffs2, che oltre alle librerie C richiede che sul sistema siano presenti le librerie condivise libz, installata per risolvere anche le dipendenze di openssl, e la libreria lzo.

Appendice B – Test

Di seguito sono riportati alcuni dei file di comandi utilizzati per eseguire i test.

Ipc:

```
#DESCRIPTION:Interprocess communication stress
# These tests use tests/pipeio to put pipes (named or unnamed) through a
workout
#
pipeio_1 pipeio -T pipeio_1 -c 5 -s 4090 -i 100 -b -f x80
# spawns 5 children to write 100 chunks of 4090 bytes to a named pipe
# using blocking I/O
#pipeio_2 pipeio -T pipeio_2 -c 5 -s 4090 -i 100 -f x80
# spawns 5 children to write 100 chunks of 4090 bytes to a named pipe
# using non-blocking I/O
# This test hits EAGAIN, which pipeio doesn't handle at the moment
pipeio_3 pipeio -T pipeio_3 -c 5 -s 4090 -i 100 -u -b -f x80
# spawns 5 children to write 100 chunks of 4090 bytes to an unnamed pipe
# using blocking I/O
pipeio_4 pipeio -T pipeio_4 -c 5 -s 4090 -i 100 -u -f x80
# spawns 5 children to write 100 chunks of 4090 bytes to an unnamed pipe
# using non-blocking I/O
pipeio_5 pipeio -T pipeio_5 -c 5 -s 5000 -i 10 -b -f x80
# spawns 5 children to write 10 chunks of 5000 bytes to a named pipe
# using blocking I/O
pipeio_6 pipeio -T pipeio_6 -c 5 -s 5000 -i 10 -b -u -f x80
# spawns 5 children to write 10 chunks of 5000 bytes to an unnamed pipe
# using blocking I/O
```

```
#pipeio_7 pipeio -T pipeio_7 -c 5 -s 5000 -i 10 -f x80
# spawns 5 children to write 10 chunks of 5000 bytes to a named pipe
# using non-blocking I/O
# This test hits EAGAIN, which pipeio doesn't handle at the moment
pipeio_8 pipeio -T pipeio_8 -c 5 -s 5000 -i 10 -u -f x80
# spawns 5 children to write 10 chunks of 5000 bytes to an unnamed pipe
# using non-blocking I/O

sem01 sem01
sem02 sem02

message_queue_test_01         message_queue_test_01
message_queue_test_02_get     message_queue_test_02_get
message_queue_test_02_snd     message_queue_test_02_snd
message_queue_test_02_rcv     message_queue_test_02_rcv
message_queue_test_02_ctl     message_queue_test_02_ctl -r
message_queue_test_04         message_queue_test_04
message_queue_test_05         message_queue_test_05
pipe_test_01                  pipe_test_01
pipe_test_02                  pipe_test_02
semaphore_test_01       run_semaphore_test_01.sh
semaphore_test_02       semaphore_test_02
semaphore_test_03       semaphore_test_03
shmem_test_01                 shmem_test_01
shmem_test_02                 shmem_test_02
shmem_test_03                 shmem_test_03
shmem_test_04                 shmem_test_04
shmem_test_05                 shmem_test_05
```

```
shmem_test_06                shmem_test_06
shmem_test_07                shmem_test_07
signal_test_01               signal_test_01
signal_test_02               signal_test_02
signal_test_03               signal_test_03
signal_test_04               signal_test_04
signal_test_05               signal_test_05
signal_test_06               signal_test_06
signal_test_07               signal_test_07
```

mm:

```
#DESCRIPTION:Memory Mgmt tests
mm01 mmap001 -m 10000
# 40 Mb mmap() test.
# Creates a 10000 page mmap, touches all of the map, sync's it, and
# munmap()s it.
mm02 mmap001
# simple mmap() test.
#mm03 mmap001 -i 0 -I 1 -m 100
# repetitive mmapping test.
# Creates a one page map repetitively for one minute.

mtest01 mtest01 -p80
mtest01w mtest01 -p80 -w

#test for race conditions
#mtest05 mmstress
mtest06 mmap1 -x 0.05
```

```
mem01 mem01

mem02 mem02


page01 page01

page02 page02


data_space data_space

stack_space stack_space


shmt02 shmt02

shmt03 shmt03

shmt04 shmt04

shmt05 shmt05

shmt06 shmt06

shmt07 shmt07

shmt08 shmt08

shmt09 shmt09

shmt10 shmt10


shm_test01  shm_test -l 10 -t 2

# mallocstress01  mallocstress
```

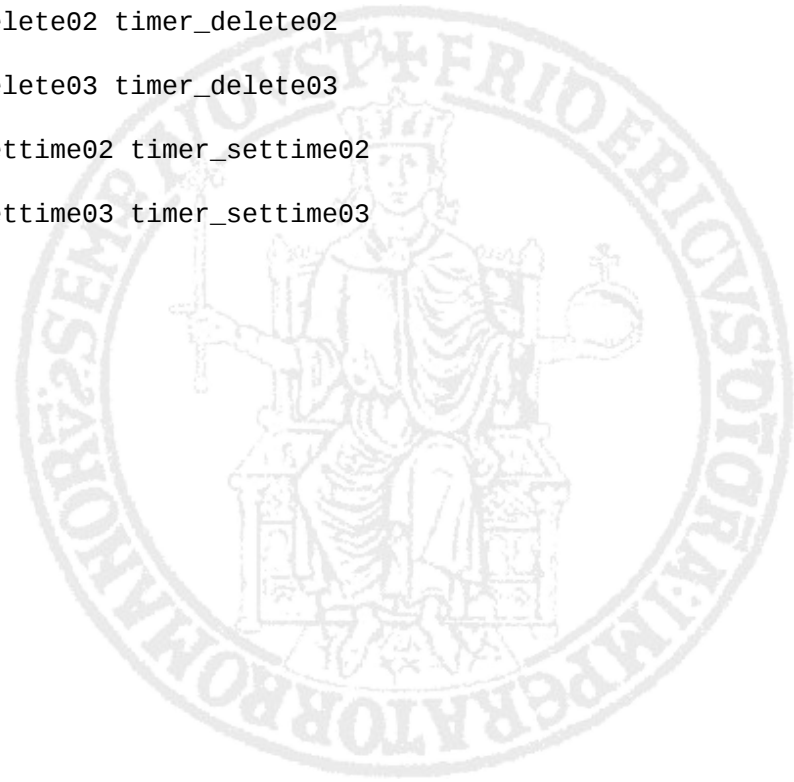
sched:

```
#DESCRIPTION:Scheduler Stress Tests

pth_str01 pth_str01

pth_str02 pth_str02 -n1000

pth_str03 pth_str03
```

```
time-schedule01         time-schedule

trace_sched01           trace_sched -c 1


hackbench01 hackbench 150 process 1000

hackbench02 hackbench 20 thread 1000
```

timers:

```
#DESCRIPTION:Posix Timer Tests

clock_gettime02 clock_gettime02

clock_gettime03 clock_gettime03

clock_settime02 clock_settime02

clock_settime03 clock_settime03

timer_create02 timer_create02

timer_create03 timer_create03

timer_create04 timer_create04

timer_delete02 timer_delete02

timer_delete03 timer_delete03

timer_settime02 timer_settime02

timer_settime03 timer_settime03
```



Bibliografia

[1] Greg Kroah-Hartman, 2006 “Linux Kernel in a Nutshell”, O'Really & Associates

[2] Kwan L. Lowe, 2004 “Kernel Rebuild Guide”, Digital Hermit

[3] Grub info pages

[4] Gerard Beekmans, 2007 “Linux from Scratch”

[5] Willian Stallings, 2005 “Operating Systems Internals and Design Principles”, Fifth editions, Prentice Hall

[6] P.Raghavan, Amol Lad, Sriram Neelakandan, 2006 “Embedded Linux System Design and Development”, Auerbach Publications

[7] Karim Yaghmour 2003 “Building Embedded Linux Systems”, O'Reilly & Associates

[8] Concurrent Technologies Inc, 2007 Techical reference Manual for VP 41x/03x VME

[9] Concurrent Technologies Inc, “Application Engineering technical reference, Booting Linux from application flash using the Linux MTD driver”

[10] Cliff Brake, feb 2006, “Tips for planning an embedded Linux project”, Embedded Linux Journal Online

[11] Richard A. Sevenich, “Retraining for Embedded Linux Development”, Embedded Linux Journal Online

[12] Joachim Henkel, Mark Tins, 2004 “Munich/MIT Survey: The Development of embedded Linux”, Embedded Linux Journal Online

[13] Victor Yodaiken, “RealTime Linux”, http://www.rtlinux.org Dipartimento di Computer Science, New Mexico Institute of Technology

[14] Edgar F. Hilton, Victor Yodaiken, 2001 “Real-Time Applications with RTLinux”, Embedded Linux Journal Online

[15] Peter Laurich, 2004 “A comparision of hard real time alternatives”, Embedded Linux Journal Online

[16] Paolo Minazzi, 2006 “Real time, la soluzione Linux (italiana) con RTAI”, Linux&C numero 54

[17] Luigi Genoni, “Intervista a Robert Love”, Linux&C numero 23

[18] Luigi Genoni, “Intervista a Ingo Molnar”, Linux&C numero 27

[19] Luigi Genoni, “Verso il Kernel 2.6”, Linux&C numero 33

[20] David Woodhouse, “JFFS: The Journaling Flash File System”, Red Hat, inc.

[21] M. Tim Jones, 2007 “Anatomy of Linux flash file systems”, IBM DeveloperWorks

[22] Cortney Jacobsen, 2006 “Selecting a flash file system for wireless devices”, Embedded Linux Journal Online

[23] Laura Cannas, Francesco Davide Carnovale, Ramona Congiu, “File system e jffs2”, Laboratorio di Progettazione di Sistemi Operativi – Università di Cagliari

[24] M. Tim Jones, 2006 “BusyBox simplifies embedded Linux System”, IBM DeveloperWorks

[25] Martin C. Brown, “Build a GCC-based cross compiler for Linux”, IBM DeveloperWorks

[26] Gary V. Vaughan, Ben Elliston, Tom Tromey, Ian Lance Taylor, 2000 “GNU autoconf, automake, and Libtool”, New Riders

[27] Luigi Genoni, 2006 “Kernel News”, Linux&C numero 64

[28] Manuale X86 di Gentoo Linux, http://www.gentoo.org/doc/it/handbook/index.xml

[29] Manuale Gentoo Embedded, http://www.gentoo.org/proj/en/base/embedded/handbook/

[30] Lina Martensson, Valerie Henson 2005 “Hacking the Linux 2.6 kernel, Part 1”, IBM DeveloperWorks

[31] Ian Shields, 2005 “Linux installation and package management”, LPI exam 101 prep, IBM DeveloperWorks

[32] Ian Shields, 2006 “Boot, Initialization, shutdown, and runlevels”, LPI exam 102 prep Topic 106, IBM DeveloperWorks

[33] Clark Williams, 2002 “Linux Scheduler Latency”, Red Hat inc, Embedded Linux

Journal Online

[34] William von Hagen, 2006 “The definitive guide to GCC, Second Edition”, Apress

[35] Claudio Panichi, 2005 “Il processo di boot: INIT, Sysctl e gestione dei runlevel”, Linux&C numero 48

[36] David Mertz, 2005 “System startup”, LPI exam 201 prep topic 202, IBM DeveloperWorks

[37] James Chapman, 2006 “Embedded Linux Best Practices”, Embedded Linux Journal Online

[38] Lars Wirzenius, Joanna Oja, Stephen Stafford, Alex Weeks, 2003 “The Linux System Administrator's Guide”

[39] Cliff Brake, Jeff Sutherland, 2001 “Flash filesystem for Embedded Linux System”, Embedded Linux Journal Online

[40] James Hunt, 2003 “How to boot Linux faster”, Embedded Linux Journal Online

[41] http://upstart.ubuntu.com

[42] Thomas Petazzoni, Karsten Kruse, Ned Ludd, Martin Herren and others, 2007 “Buildroot”, http://buildroot.uclibc.org/buildroot.html

[43] Greg O' Keefe, 2000 “From power up to bash prompt”

[44] Gilard Ben-Yossef, “Benchmarking boot latency on x86”, Embedded Linux Journal Online

[45] http://www.bootchart.org/

[46] Subrata Modak, Balbir Singh, “Building a robust Linux kernel piggybacking the Linux Test Project”, Linux Technology Centre, IBM, India

[47] Niget Hinds, Paul Larson, Hubertus Franke, Marty Ridgeway, “Using code coverage tools in the Linux kernel”, IBM

[48] Robert Williamson, 2004 “Stress-testing the Linux Kernel”, IBM DeveloperWorks

[49] Roger S. Pressman, 2000 “Principi di Ingegneria del software”, terza edizione McGraw Hill

[50] Kevin Dankwardt, 2002 “Real-Time and Linux, part 1”, Embedded Linux Journal

Online

[51] Kevin Dankwardt, “Real-Time and Linux, part 2: the Preemtible kernel”, Embedded Linux Journal Online

[52] Kevin Dankwardt, “Real-Time and Linux, part 3: Sub-kernels and Benchmarks”, Embedded Linux Journal Online

[53] Brandon White, 2003 “A Breakthrough for Embedded Systems”, Embedded Linux Journal Online

[54] 2000 “MontaVista unveils fully preemptable Linux kernel prototype”, LinuxDevices.com (http://www.linuxdevices.com/news/NS7572420206.html)

[55] Judy DeMocker, 2001 “Three reasons why Linux will trounce the embedded market”, IBM DeveloperWorks

[56] Eric Steven Raymond, 1998 “Colonizzare la noosfera”, Apogeonline http://www.apogeonline.com/openpress/homesteading